Improving the implementation of traits in Pharo

Sebastián Tleye

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Improving the implementation of traits in Pharo

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Sebastián Tleye

Director: Damien Cassou

Codirector: Hernán Wilkinson

Buenos Aires, 2013

RESUMEN

Los traits son un nuevo concepto en la programación orientada a objetos, fueron incluidos en lenguajes de programación como PHP, Perl y Pharo/Smalltalk. Son unidades puras de comportamiento que pueden ser compuestas para formar clases u otros traits. El mecanismo de composición de los traits es una alternativa a la herencia múltiple o por mixins en la cual el compositor tiene un completo control sobre la composición del trait. El resultado permite mas reúso que la herencia simple sin introducir los inconvenientes de la herencia múltiple o por mixins.

Pharo es un nuevo ambiente de desarrollo open-source inspirado en Smalltalk. Provee una plataforma innovativa de desarrollo tanto para la industria como para la investigación. Los traits están integrados en Pharo pero raramente usados debido a la falta de soporte en el ambiente de desarrollo.

Por esto, es necesario crear más soporte a los programadores en el uso de traits. Hacer que las clases y los traits sean polimórficos entre si es una buena forma de empezar. Todas las herramientas que saben como manejar clases sabrán también como manejar traits. Lograr el máximo polimorfismo posible entre clases y traits es uno de los objetivos de este trabajo.

Arreglar bugs presentes en Pharo también forma parte del trabajo. Hay bugs en Pharo que están presentes desde las primeras implementaciones de traits. Se van a arreglar estos bugs y se va a explicar por qué son importantes.

Finalmente, vamos a pensar una nueva forma de visualizar traits, y con esto, abrir nuevas puertas para que futuros programadores puedan implementarla.

Palabras claves: Traits, Pharo, Smalltalk, POO, Refactoring, Polimorfismo.

ABSTRACT

Traits are a new concept in Object-Oriented Programming, they were included in programming languages such as PHP, Perl and Pharo/Smalltalk. Traits are pure units of behavior that can be composed to form classes or other traits. The trait composition mechanism is an alternative to multiple or mixin inheritance in which the composer has full control over the trait composition. The result enables more reuse than single inheritance without introducing the drawbacks of multiple or mixin inheritance.

Pharo is a new open-source Smalltalk-inspired programming language and environment. It provides a platform for innovative development both in industry and research. Traits are integrated in Pharo but barely used due to a lack of support in the development environment.

For this reason, it is necessary to create more support to programmers when using traits. Making classes and traits polymorphic is a good start. All the tools that know how to handle classes will also know how to handle traits. Achieve the maximum possible polymorphism between classes and traits is one of the goals of this work.

Fix bugs in Pharo is also part of this work. There are bugs in Pharo presents since the first implementations of traits. We will fix these bugs and we will explain why they are important.

Finally, we will think a new way to visualize traits, and with this, open doors to future programmers to implement it.

Keywords: Traits, Pharo, Smalltalk, OOP, Refactoring, Polymorphism.

ACKNOWLEDGMENT

Quiero agradecer a Damien Cassou por haber dirigido la tesis y por su ayuda constante. A Stéphane, Inria y todo el grupo RMoD por la buena onda y por haberme posibilitado hacer la tesis con ellos. A Hernán Wilkinson por aceptar co-dirigirla.

A mis profesores del secundario que me introdujeron en el mundo de la programación.

Agradezco a mi familia, estudiar hubiera sido imposible sin su ayuda. A mi abuela, que me ayudó en los momentos economicamente mas dificiles. A Mariela, por soportarme durante el trayecto final de mi carrera.

A mis amigos, con los que estudiar se hizo mas fácil y divertido. A Juanma, gracias a él hice la pasantía en Inria. A mi compa remero, con él estudiamos para los finales mas dificiles.

Y finalmente quiero agradecer a la educación pública, a la que le debo la mayor parte de mi formación.

Índice general

1. INTRODUCTION

Inheritance is well-known and accepted as a mechanism for reuse in object-oriented languages. Unfortunately, due to the coarse granularity of inheritance, it may be difficult to decompose an application into an optimal class hierarchy that maximizes software reuse. Existing schemes based on single inheritance (e.g., in Java [1]), multiple inheritance (e.g., in C++ [2]), and mixin (e.g., in Ruby [3]), all pose numerous problems for reuse.

To overcome these problems, a new composition mechanism called Traits [4], has been included in programming languages such as PHP [5], Perl [6] and Pharo/Smalltalk [7, 8]. Traits are pure units of behavior that can be composed to form classes and other traits. The trait composition mechanism is an alternative to multiple and mixin [9] inheritance in which the composer has full control over the trait composition. The result enables more reuse than single inheritance without introducing the drawbacks of multiple and mixin inheritance.

Pharo is a new open-source Smalltalk-inspired programming language and environment. It provides a platform for innovative development both in industry and research. Traits are integrated in Pharo but barely used due to a lack of support in the development environment.

One of this missing feature you can see it in tools. Tools used today in Smalltalk were not designed to deal with traits, as a consequence it is not strange to see bugs and ugly code in tools. One of the things we did is to make classes and traits polymorphic between them. This will make that tools that know how to handle classes, will also know how to handle traits.

After making classes and traits polymorphic, we will import to Pharo an implementation for refactorings made by Alejandro Gonzalez [10] in Squeak. Doing this, we will not only test the system but we will also provide Pharo an important missing feature.

Since the first implementations of traits, Pharo has had significant bugs, this generates refusal by programmers when using traits. Fix some of these bugs is also part of this thesis. We will explain and fix two important bugs present since the first implementations of traits in Squeak/Pharo.

Nautilus, the current browser of classes in Pharo, is based on the old class browser of Smalltalk, and was not thought to handle traits. Therefore, classes and traits are shown equally, generating a mix of classes, traits and methods in the browser that can be confusing and upsetting. We will conclude this work by thinking a new way to visualize traits, and with this, open doors to future programmers to implement it.

1.1. Structure of the thesis

Chapter 2 introduces the notion of trait. Also, we will present the problems of the hierarchies used so far and how they are solved using traits.

Chapter 3 explains the current hierarchy of classes used to model traits. The similarities and differences between the hierarchy of classes and the hierarchy of traits. How polymorphism between classes and traits was achieved, and the benefits/problems that this brings. We will show also how the polymorphism is kept after the integration.

Chapter 4 talks about refactorings for traits. These refactorings were made in Squeak by Alejandro Gonzalez, we will import them to Pharo. Doing this we will put on test the system previously changed and we will also provide to Pharo a missing feature.

Chapter 5 and 6 show some important bugs in Pharo related to traits. Why is needed to fix them and how they were fixed, bugs that were present since the first implementations of traits.

Chapter 7 and 8, close this work with conclusions and talking about possible future works.

1.2. Background

Chapter 4 is based on the master thesis made by Alejandro Gonzalez titled "Mejorando la utilidad de las herramientas de refactorings". He modified the refactoring model in Squeak [11] and then he added some refactorings for traits. Part of the work of this thesis consisted in taking the model implemented by him and ported to Pharo.

Regarding future work, we took as a base the objects visualizer Gaucho. The idea was to add traits and then turn that into a browser. This is in the future work chapter since the implementation was never made.

1.3. Implementation

The development environment we used is Pharo. Pharo is an open-source Smalltalk inspired programming language and environment. Pharo contains only code that has been contributed under the MIT [12] license.

2. TRAITS

Although single inheritance is widely accepted in Object-Oriented programming, programmers have long realized that single inheritance is not expressive enough to factor out common features shared by classes in a complex hierarchy. As a consequence, language designers have proposed various forms of multiple inheritance, as well as other mechanisms such as mixins, that allow classes to be composed incrementally from sets of features.

2.1. Single inheritance

Single inheritance is the simplest model of inheritance; it allows a class to inherit from at most one superclass. Although this model is well-accepted, it is not expressive enough to allow the programmer to factor out all the common features shared by classes in a complex hierarchy.

As an example we can think on the `Stream` hierarchy of classes in Smalltalk. Figure 2.1.

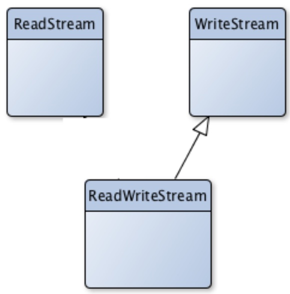


Fig. 2.1: Hierarchy of Stream in Smalltalk

We have the entities `ReadStream, WriteStream` and `ReadWriteStream`. As it name suggest, `ReadWriteStream` has the behavior provided by `ReadStream` and `WriteStream`. However, the single inheritance model allows to `ReadWriteStream` inherits only from one of these classes. In Smalltalk, `ReadWriteStream` inherits from `WriteStream` and the behavior of `ReadStream` is copied in `ReadWriteStream`.

2.2. Multiple inheritance

In this kind of hierarchies, a class can inherit from more than one superclass. This has the benefits of better reuse of the code and a more flexible model.

The previous hierarchy could be easily solved using multiple inheritance. See Figure **??**

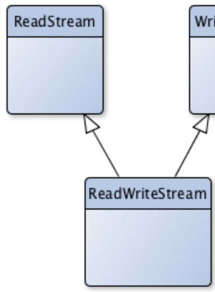


Fig. 2.2: Hierarchy of Stream using multiple inheritance

Multiple inheritance brings some problems, including the known “diamond problem”, that occurs when a class inherits the same base class through different paths. It is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and/or C has overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?

2.3. Mixins

A mixin is a class that offers certain functionality to be inherited by a subclass, but it is not thought to be autonomous. Inheriting from a mixin is not a way of specialization but a way to obtain functionality. Mixins allow the programmer to achieve better code reuse than single inheritance while maintaining the simplicity of the inheritance operation. However, mixins have limitations when you want to compose more than one mixin for the same class.

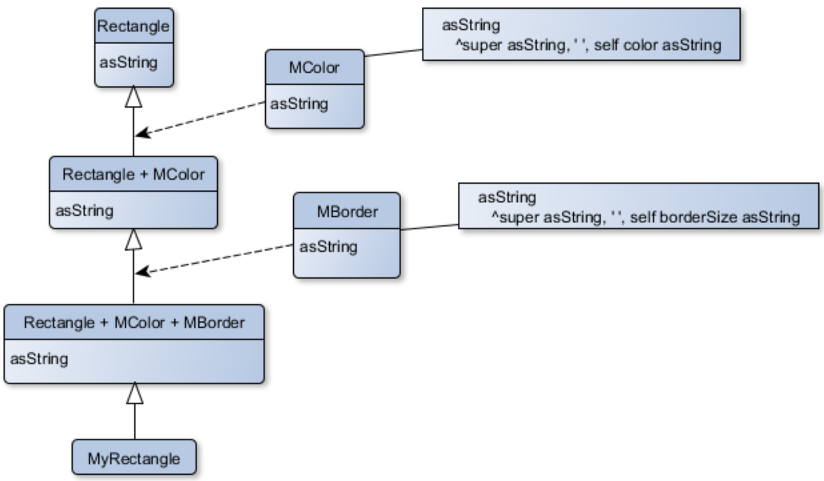


Fig. 2.3: Example of mixins

Figure 2.3 shows that the composed class `MyRectangle` cannot access the implementation of `asString` in the mixin `MColor` nor the class `Rectangle`. Classes `Rectangle + MColor` and `Rectangle + MColor + MBorder` are generated by the application of mixins.

Also, mixins generate fragile hierarchies. Due the linearity and lack of resources to solve conflicts, when multiple mixins are composed you get as a result a fragile hierarchy, not very flexible about changes. Adding a new method to one of the mixins may silently override an identically named method of a mixin that appears earlier in the chain.

2.4. Traits

A trait is essentially a group of methods that serves as building blocks for classes and is a primitive unit of code reuse. Classes are composed from a set of traits by specifying *glue code* that connects the traits together and accesses the necessary state.

A trait contains a set of methods that implements the behavior that it *provides*. In general, a trait may *require* a set of methods that serves as parameters for the provided behavior. Traits cannot specify any state, and never access state directly. `Trait` methods can access state indirectly, using required methods that are ultimately satisfied by accessors (getter and setter methods).

The purpose of traits is to decompose classes into reusable building blocks by providing first-class representations for the different aspects of the behavior of a class.

Traits have the following properties.

- A trait provides a set of methods that implements behavior;
- A trait requires a set of methods that serves as a parameter for the provided behavior;
- Traits do not specify any state variables, and the methods provided by traits never access state variables directly;
- Classes and traits can be composed from other traits, and the composition order is irrelevant. Conflicting methods must be *explicitly* resolved;
- Trait composition does not affect the semantics of a class: the meaning of the class is the same as it would be if all of the methods obtained from the trait(s) were defined directly in the class;
- Similarly, trait composition does not affect the semantics of a trait: a composite trait is equivalent to a *flattened* trait containing the same methods.

2.4.1. Example

Figure 2.4 shows how the class `Rectangle` is using the trait `TMovable` and `TResizable`. `TMovable` is a trait that represents an object that can be moved, it requires setters for the coordinates `x` and `y`. `TResizable` is a trait that represents an object that can change its size. It provides the method `resizeWidht:andHeight:` which changes the size of the user, and requires setters for `width` and `height`.

Each instance of `Rectangle` knows how to answer the messages `x:`, `y:`, `width:`, `height:`, `moveToX:Y:` and `resizeWidth:andHeight:`

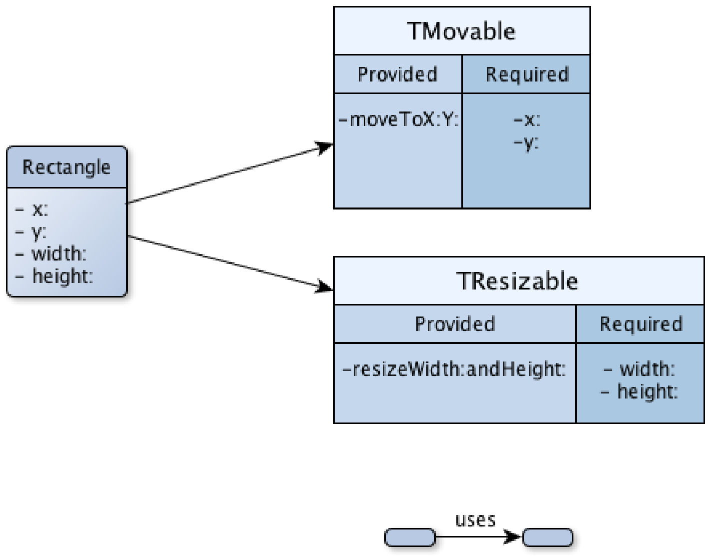


Fig. 2.4: Class Rectangle using the traits TMovable and TResizable

Let's see the definition of the class and the two traits.

```
Object subclass: #Rectangle
   uses: TMovable + TResizable
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'CategoryName'

Rectangle>>x: aValue
   x := aValue

Rectangle>>y: aValue
   y := aValue

Rectangle>>width: aValue
   width := aValue

Rectangle>>height: aValue
   height := aValue
```

```
Trait named: #TMovable
   uses:
   category: 'CategoryName'

TMovable>>moveToX:anXValue Y:anYValue
   self x: anXValue.
   self y: anYValue.

TMovable>>x:anXValue
   ^self explicitRequirement.

TMovable>>y:anYValue
   ^self explicitRequirement.
```

```
Trait named: #TResizable
   uses:
   category: 'CategoryName'

TResizable>>resizeWidth:w andHeight:h
   self width: w.
   self height: h.

TResizable>>width:aWidth
   ^self explicitRequirement.

TResizable>>height:anHeight
   ^self explicitRequirement.
```

`explicitRequirement` is the way to say “this method is a requirement”

Then, the hierarchy of section 2.1 can be easily solved implementing the repeated methods in the traits.

2.4.2. Trait compositions

Traits can be composed to form a trait composition. Then, a class (or other trait) can use this trait composition and have all its methods.

A trait composition has the flattening property, which says that the semantics of a method defined in a trait is identical to the semantics of the same method defined in a class that uses the trait.

Traits compositions are formed by transformations, a transformation can be either a trait, an exclusion or an alias (T_i, -, @). In chapter 6, we will give a deeper view on trait compositions.

3. POLYMORPHISM BETWEEN CLASSES AND TRAITS

3.1. Introduction

Traits are a relative new concept in Object-Oriented Programming, they were not implemented in the old Smalltalk-80 but introduced later in Squeak 3.9 [11] on 2008. Thus, some of the tools used today in Smalltalk, were not designed to deal with them (e.g. Monticello).

Tools had to be modified in order to support traits, this is the case of the class browser for example, in which traits need to be shown together with classes. Also, facilities like browsing the users of a trait, or differentiating methods coming from a trait to the methods of the class had to be added.

Programmers tried to make traits “more polymorphic” with classes by adding methods like `superclass` or `subclasses` to the class `Trait`. But despite of this, there is still code in Pharo in which a difference between classes and traits is needed. This is the case of Nautilus for example, in which some methods have code like this: `cls isTrait ifTrue:[do something] ifFalse:[do a different thing]`, is for this reason that making classes and traits polymorphic is needed.

Let’s see a piece of code taken from Pharo

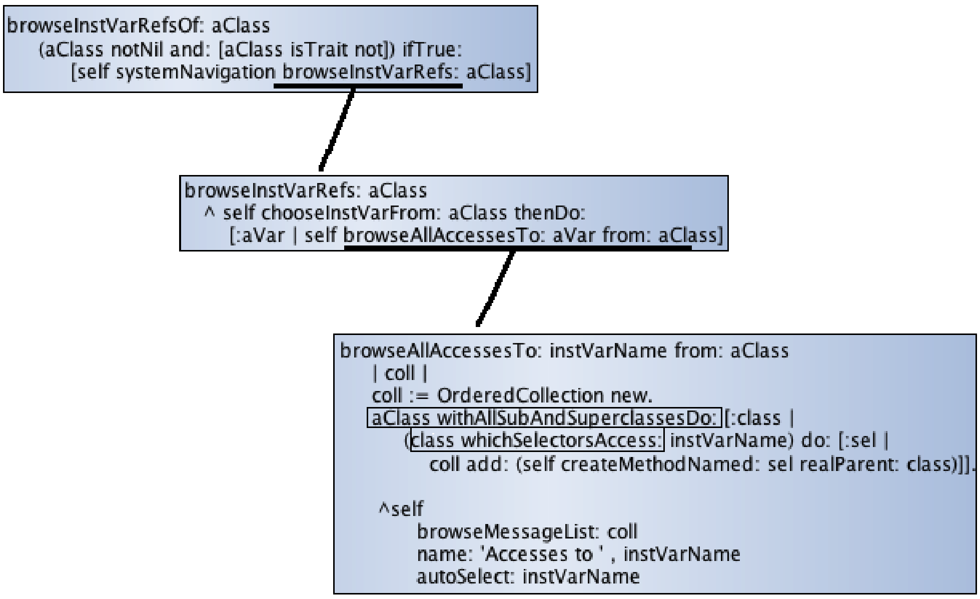


Fig. 3.1: Code taken from Pharo

As you can see, the method `browseInstVarRefsOf:` checks if `aClass` is or not a trait. This is so because as you can see after, the messages `withAllSubAndSuperclassesDo:` and `whichSelectorsAccess:` are sent to `aClass` and since these messages are not implemented in `Trait` neither `TraitDescription` nor `TraitBehavior`, then traits do not know how to answer them.

If we want to remove that "if", then `aClass` should know how to answer the messages `withAllSubAndSuperclassesDo:` and `whichSelectorsAccess:`. Making classes and traits polymorphic will not only solve problems like this, but will also decrease the chance of bugs that are not known yet.

3.2. Model of classes in Smalltalk

In Smalltalk everything is an object, and every object is an instance of a class. Classes are no exception: classes are objects, and class objects are instances of other objects. This object model captures the essence of object-oriented programming: it is lean, simple, elegant and uniform.

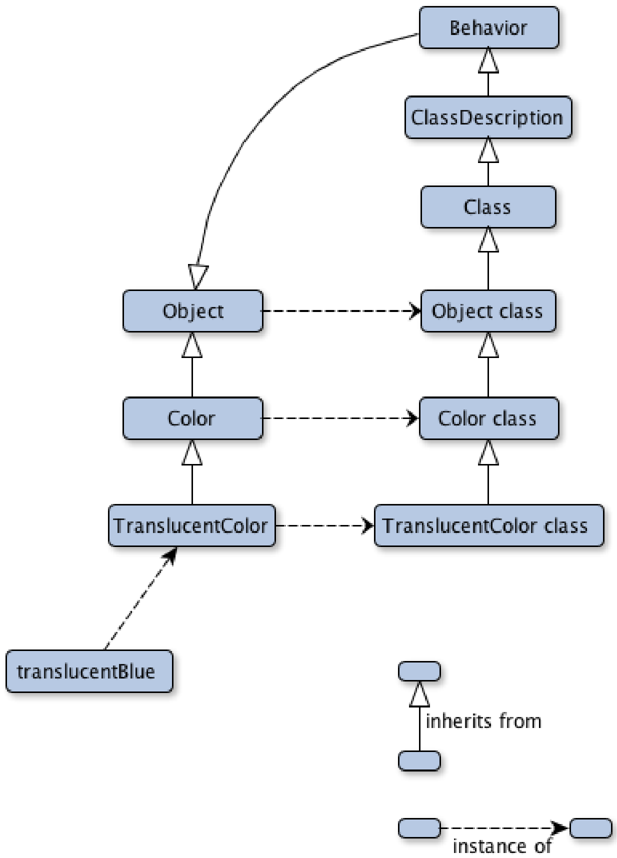


Fig. 3.2: Metaclasses inherit from `Class` and `Behavior`

Since `classes` are objects, and `every object is an instance of a class`, it follows that classes must also be instances of classes. A class whose instances are classes is called `metaclass`. Whenever a class is created, the system automatically creates a metaclass. A metaclass defines the behavior of certain classes and their instances.

The classes to model classes in Smalltalk are `Behavior`, `ClassDescription`, `Class` and `Metaclass`.

`Behavior` provides the minimum state necessary for objects that have instances: this includes a superclass link, a method dictionary and a description of the instances. `Behavior` inherits from `Object`, so instances of it, and its subclasses, can behave like objects.

`ClassDescription` is an abstract class that provides facilities needed by its two direct subclasses, `Class` and `Metaclass`. `ClassDescription` adds a number of facilities to the basis provided by `Behavior`: named instance variables, the categorization of methods into protocols, the maintenance of `change sets`, logging of changes, and most of the mechanisms needed for filing-out changes.

`Class` represents the common behavior of all classes. It provides a class name, compilation methods, method storage, and instance variables. It provides a concrete representation for class variable names and shared pool variables (`addClassVarNamed:`, `addSharedPool:`).

`Metaclass` represents common metaclass behavior. It provides accessors to its instance which is unique, and some behavior shared with `Class`.

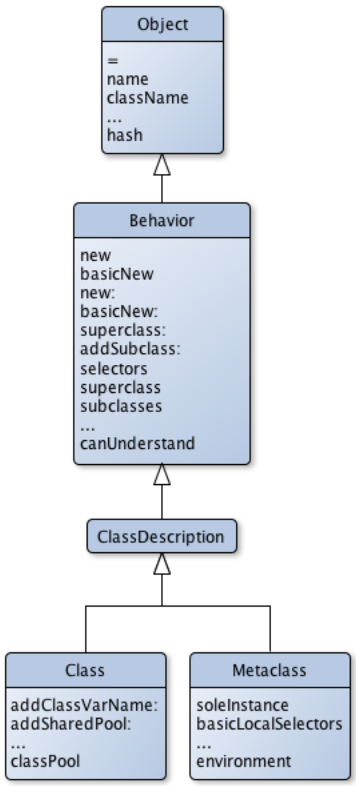


Fig. 3.3: Kernel classes in Smalltalk

3.3. Model of traits in Smalltalk

Traits are similar concept to classes, both define a behavior that is shared by its instances (in the case of classes) or users (in the case of traits), although traits have no instances they also have a behavior class which is called `TraitBehavior`. `TraitBehavior` provides the minimum state necessary for all the traits: this includes users manipulation, traits used, and a large number of shared methods with classes.

`TraitDescription`, analogously to `ClassDescription`, is an abstract class that provides facilities for both of its subclasses, `Trait` and `ClassTrait`. Some of them are:

- Category organization for methods
- Maintenance of a ChangeSet
- Logging changes on a file
- Mechanism for fileOut
- Copying of methods to other traits/classes

`TraitDescription` also provides composition methods: `-, +, @`

Each trait in the system is represented as an instance of `Trait`. Like `Class`, `Trait` concretizes `TraitDescription` by providing instance variables for the name and the environment. Since traits do not define variables, `Trait` do not provide facilities for pool variables, however, it declares an instance variable `classTrait` to hold the associated classTrait, which is the "metaclass" of the trait.

While every class has an associated metaclass, a trait can have an associated classTrait, an instance of `ClassTrait`. To preserve metaclass compatibility, the associated classTrait is automatically applied to the metaclass, whenever a trait is applied to a class.

In Figure 3.4 you can see how the class `Behavior` uses the trait `TBehavior` which is an instance of `TBehavior classTrait`, and `Behavior` is an instance of `Behavior class` which uses the trait `TBehavior classTrait`.

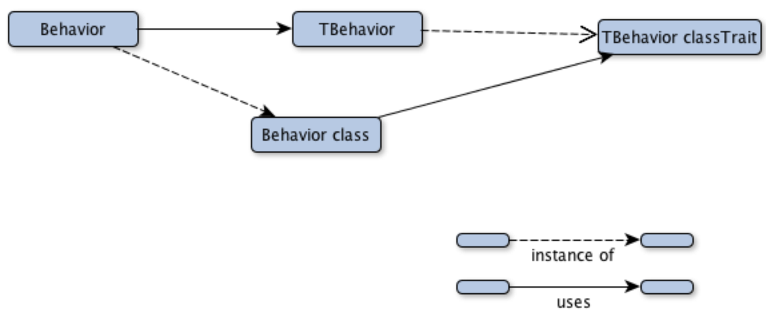


Fig. 3.4: Class side of traits

Being more technical, all the traits in the system are not instances of its classTrait but instances of `Trait`, to get the classTrait of a trait you need to ask for the instance variable `classTrait`. See Figure 3.5

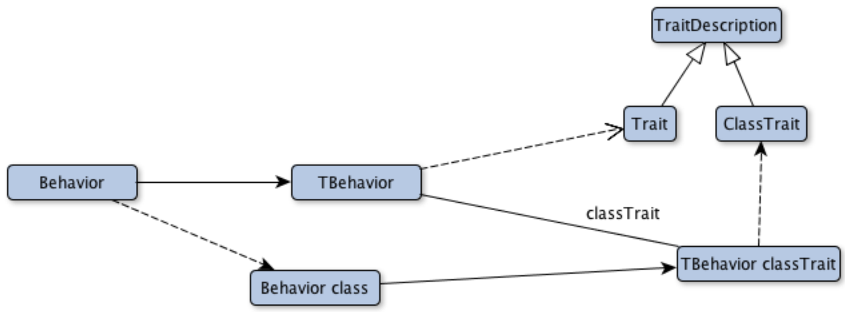


Fig. 3.5: Class side of traits

Finally, the hierarchy of the kernel classes of traits is shown in Figure3.6, notice how this hierarchy of traits is symmetric to the hierarchy of classes.

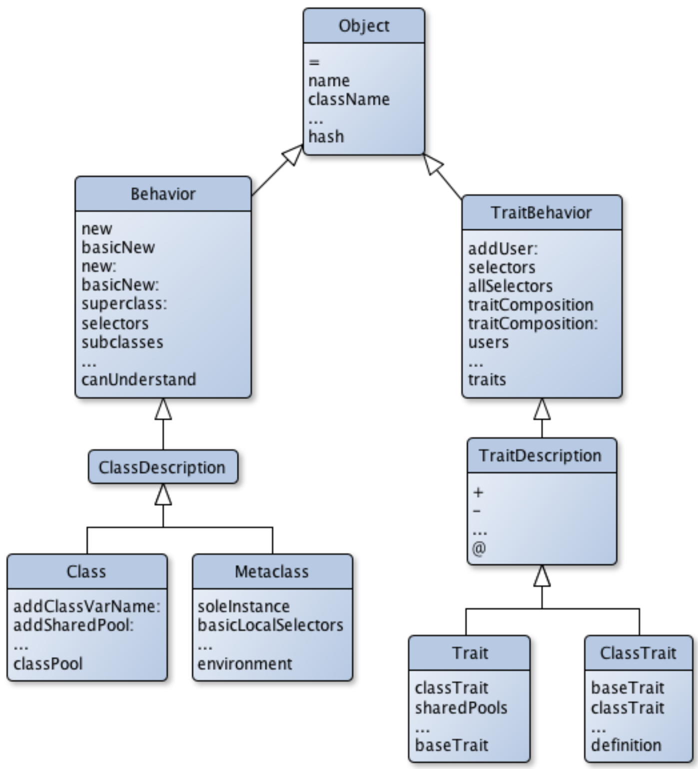


Fig. 3.6: Kernel trait classes in Smalltalk

It is worth saying that the class `TraitDescription` was created to have a parallel hierarchy with classes, but both `ClassDescription` and `TraitDescription` are not well defined classes. If you look the implementation and the methods of these classes, you realized that this behavior can be implemented somewhere else, for example in their superclasses. Chapter 7 shows a possible modification to the hierarchy.

3.4. Shared methods between classes and traits

In Pharo, `Behavior` and `TraitBehavior` share a large number of methods. It is because of this that those methods are implemented in a trait called `TPureBehavior`, the same applies to `ClassDescription` and `TraitDescription`, there is a trait called `TClassAndTraitDescription` which implements repeated methods (`Trait/Class` and `Metaclass/ClassTrait` also have a common trait but with a few methods only).

In Figure 3.7 (Roassal graphic [13]) we can see hierarchy of classes and traits in Pharo. Each box represents either a trait or a class, the height of the box represents the amount of local methods in the class (i.e. without including trait methods) and the width represents the number of variables.

`Behavior` and `TraitBehavior` use the trait `TPureBehavior`. As a result they share 110 methods. `ClassDescription` and `TraitDescription` use the trait `TClassAndTraitDescription` and they share 75 methods. Finally, `Metaclass` and `ClassTrait` are sharing 2 methods in the trait `TApplyingOnClassSide`.

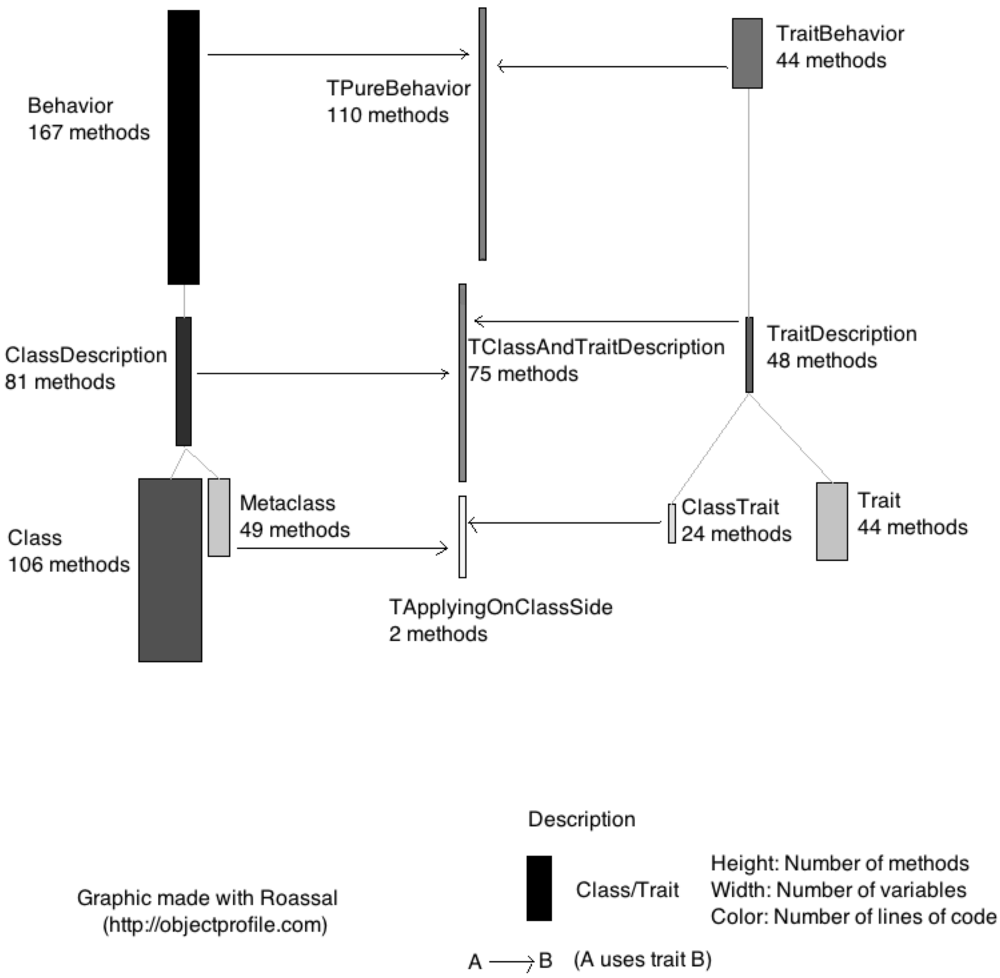


Fig. 3.7: Kernel classes and traits before polymorphism

3.5. Sharing more methods

One way to make classes and traits "more polymorphic" is to move more methods to the traits, this would be to move more methods from `Behavior` and `TraitBehavior` to `TPureBehavior`, from `ClassDescription` and `TraitDescription` to `TClassAndTrait-Description`, from `Metaclass` and `ClassTrait` to `TApplyingOnClassSide`, and from `Class` and `Trait` to `TBehaviorCategorization`.

This is not a trivial task, it is not a matter of copying and pasting methods. For example, all the methods that directly access instance variables cannot be moved to the trait, because traits do not allow instance variables. Instance variables need to be abstracted before being moved.

Another problem is that some methods that belong to `Class` can make no sense for `Trait` or vice versa. Classes know how to answer messages that conceptually should not belong to traits, and the same the other way around. (e.g Classes know how to answer `instanceVariables` but traits do not. Traits know how to answer `users` but classes do not).

Consider the method `superclass` implemented in `Behavior`

```
Behavior>>superclass
  ^superclass
```

If we copy and paste the method to `TPureBehavior`, the trait will not know what the instance variable `superclass` is. In addition, what is conceptually the superclass of a trait?

To solve the first problem, all the accessors will stay where they are, they cannot be put in the trait, at least not in stateless traits. To solve the second problem, we have to think that if we want polymorphism between classes and traits, traits and classes must know how to answer "senseless" messages. Then, the superclass of a trait could be `nil`.

3.5.1. Methods that cannot be moved

There are methods that cannot be moved to the shared traits, this is so because they have different implementations. Some of these methods were already implemented. And some of those methods were implemented only for classes or traits but not both.

Let's see some of these methods:

Behavior>>superclass
  ^superclass
	TraitBehavior>>superclass
	  ^ nil

`superclass` was implemented only for classes. Now it is implemented also for traits and returns `nil`.

Behavior>>users
  ^IdentitySet new.
	TraitBehavior>>users
	  ^users

`users` was implemented only for traits. Now we also implemented it for classes and returns an empty collection.

Behavior>>users: aCollection
  "Compatibility purposes"
	TraitBehavior>>users: aCollection
	  users := aCollection.

The setter of `users` was implemented only for traits. Now it is implemented also in classes and it does nothing.

```
Behavior>>instSize
  self flag: #instSizeChange.
  ^ ((self format bitShift: -10) bitAnd: 16rC0) + ((self format bitShift: -1) bitAnd: 16r3F) - 1
```

```
TraitBehavior>>instSize
  ^0
```

`instSize` returns a value that depends on the instance variables of the class. Since traits have no instance variables, it returns 0 (which is the number returned for classes without instance variables).

```
Behavior>>addInstVarNamed: aString
  self subclassResponsibility
```

```
TraitBehavior>>addInstVarNamed: aString
  "Compatibility purposes"
```

`addInstVarNamed:` has different implementations for classes and metaclasses. It is because of this that the method in `Behavior` has the `subclassResposability`.

For `TraitBehavior` it does nothing since traits have no instance variables.

```
Behavior>>instanceVariables
  ^instanceVariables.
```

```
TraitBehavior>>instanceVariables
  ^#().
```

`instanceVariables` was implemented only for classes, it returns the instance variables of a class. We added the implementation for traits which returns an empty collection.

```
Class>>subclasses
  ^subclasses == nil
    ifTrue: [#()]
    ifFalse: [subclasses copy]
```

```
Trait>>subclasses
  ^ #()
```

`subclasses` returns an empty collection for traits since they have no subclasses. For classes it returns the subclasses.

```
Class>>subclasses: aCollection
  subclasses := aCollection.
```

```
Trait>>subclasses: aCollection
  "Compatibility purposes"
```

This method is the setter of subclasses. For traits it does nothing since they have no subclasses.

```
Class>>sharedPools
  sharedPools == nil
     ifTrue: [^sharedPools := OrderedCollection new].
  ^sharedPools.
```

```
Trait>>sharedPools
  ^ OrderedCollection new
```

`sharedPools` returns the shared pools of the receiver. For traits it returns an empty collection.

```
Class>>sharedPools: aCollection
  sharedPools := aCollection
```

```
Trait>>sharedPools: aCollection
  "Compatibilty purposes"
```

`sharedPools:` is the setter of shared pools. For traits it does nothing.

```
Class>>classPool
  classPool == nil
    ifTrue: [classPool := Dictionary new].

  ^classPool.
```

```
Trait>>classPool
  ^ Dictionary new
```

`classPool` returns a dictionary with the class variables. Since traits have no class variables it returns an empty dictionary.

```
Class>>classPool: aDictionary
  classPool := aDictionary
```

```
Trait>>classPool: aDictionary
  "Compatibility purposes"
```

`classPool:` is the setter of the class variables dictionary. For traits does nothing.

```
Class>>copy
  | newClass |
  newClass := self class copy new
    superclass: superclass
    methodDict: self methodDict copy
    format: format
    name: name
    organization: self organization copy
    instVarNames: instanceVariables copy
    classPool: classPool copy
    sharedPools: sharedPools copy.
  Class instSize+1 to: self class instSize do:
       [:offset | newClass instVarAt: offset put: (self instVarAt: offset)].
  ^ newClass
```

```
Trait>>copy
  | newTrait |
  newTrait := self class basicNew initialize
    name: self name
      traitComposition: self traitComposition copyTraitExpression
    methodDict: self methodDict copy
    localSelectors: self localSelectors copy
    organization: self organization copy.

  newTrait environment: self environment.
  newTrait classTrait initializeFrom: self classTrait.
  ^newTrait
```

The implementation of `copy` for classes and traits is quite different. It will not be moved to a shared trait.

```
Behavior>>basicNew
  < primitive: 70 >
  self isVariable ifTrue: [ ^ self basicNew: 0 ].
  OutOfMemory signal.
  ^ self basicNew
```

```
TraitBehavior>>basicNew
  self error: 'Traits cannot create instances'.
```

`basicNew` is similar to the previous example, the implementation is different for trait and classes.

3.5.2. Moved methods

Now, let's see some of the methods moved to the shared traits.

Before moving

```
classDepth
  superclass ifNil: [^1].
  ^superclass classDepth + 1
```

After moving

```
classDepth
  self superclass ifNil: [^1].
  ^self superclass classDepth + 1
```

`classDepth` was implemented in `Behavior` (only for classes), now is implemented in `TPureBehavior`[1] (shared between classes and traits). It returns 1 for traits since `superclass` always returns `nil`. As you can see, the instance variable superclass was abstracted to an accessor method.

Before moving

```
allSuperclasses
  | temp |
  ^ superclass == nil
    ifTrue: [ OrderedCollection new]
        ifFalse: [temp := superclass allSuperclasses.
      temp addFirst: superclass.
      temp]
```

[1] We will see that the real trait is TBehavior, not TPureBehavior

After moving

```
allSuperclasses
  | temp |
  ^ self superclass == nil
    ifTrue: [ OrderedCollection new]
      ifFalse: [temp := self superclass allSuperclasses.
    temp addFirst: self superclass.
    temp]
```

This is a similar example, the instance variable `superclass` was replaced by an accessor implemented in the user side.

I tried to make the minimum needed changes, but as you can see, code like "`(self superclass == nil) ifTrue:`" can be replaced by something more descriptive like "`self hasSuperclass ifFalse:`"

3.5.3. Merged methods

There were also methods with different implementations for classes and traits, but whose implementations can be merged.

```
Behavior>>allSubclassesDo: aBlock
  self subclassesDo:
    [:cl |
    aBlock value: cl.
    cl allSubclassesDo: aBlock]
```

```
TraitBehavior>>allSubclassesDo: aBlock
  "Do nothing"
```

As you can see, the implementation for classes and traits was different. Now, we merged the implementation in `TBehavior>>allSubclassesDo:`

```
TBehavior>>allSubclassesDo: aBlock
   self subclassesDo:
    [:cl |
    aBlock value: cl.
    cl allSubclassesDo: aBlock]
```

Notice that the merge is the same that the implementation for classes. In the case of traits, they do not have subclasses, so the block is not evaluated.

Let's see another example

```
Behavior>>lookupSelector: selector
  | lookupClass |
  lookupClass := self.
  [lookupClass == nil]
    whileFalse:
     [(lookupClass includesSelector: selector)
        ifTrue: [^ lookupClass compiledMethodAt: selector].
     lookupClass := lookupClass superclass].
  ^ nil
```

```
TraitBehavior>>lookupSelector: selector
  ^(self includesSelector: selector)
    ifTrue: [self compiledMethodAt: selector]
    ifFalse: [nil]
```

This method can be merged in one single method as we can see below.

```
TBehavior>>lookupSelector: selector
  | lookupClass |
  lookupClass := self.
  [lookupClass == nil]
    whileFalse:
      [(lookupClass includesSelector: selector)
        ifTrue: [^ lookupClass compiledMethodAt: selector].
      lookupClass := lookupClass superclass].
  ^ nil
```

It is important to note that the merge of methods makes the system slower, this is so because in general the merged method contains more messages sent. For example, the method `allSubclasses:` that we saw before had no code for traits. Then, after the merge, there are block evaluations, and other messages.

Even though there are more messages sent, it is also true that the merge of methods maximizes the reuse of code and makes classes cleaner. This is important because the shared traits can be reused later by other classes.

3.5.4. Methods that are not polymorphic

Although there are more shared methods between classes and traits, it is also important to mention that classes and traits are not completely polymorphic.

Let's see some of these methods

`format` is a method that only classes know how to answer. It returns a number that the VM uses for internal purposes. Traits do not answer this message.

`layout` is only answered by classes. It is also used by the VM for internal purposes, traits do not answer this message.

Compositions messages are only answered by traits. These methods are `+`, `@` and `-`.

These methods are not polymorphic for two reasons. First, it is difficult to find an equivalent one. What should `format` return for traits?

Second, it is strange that a tool needs to send one of these messages.

3.6. Implementation challenge

In my final implementation, you will see that the trait used by `Behavior` and `TraitBehavior` is not `TPureBehavior` but `TBehavior`, the trait used by `ClassDescription` and `TraitDescription` is not `TClassAndTraitDescription` but `TClassDescription` and the one used in `Class` and `Trait` is not `TBehaviorCategorization` but `TClass`. This is so because the implementation was made step by step in order to not break anything. Those traits were created and then used to replace the old ones.

The first thing we did was create the trait `TBehavior`. Then we copied all the methods from `Behavior` to `TBehavior` and finally we modified those methods as we explained before (changing instance variables by accessors, merging, etc). Once we copied all the methods to the trait, we changed the definition of the class `Behavior` to use the trait `TBehavior`.

One of the features of Smalltalk (and Pharo in particular) is that while we are programming, we are living in a world of live objects, not in a world of static program text. All the changes we do affect the system, and touching kernel classes can make the system crash. As a consequence, making a kernel class use a trait is not as easy as making any other class using a trait.

When we make a class using a trait, there are compiling and other methods that are called, and some of those methods are implemented in the kernel hierarchy of classes, then if we change the trait used by a kernel class, we lose the link to the old trait, in which some of the needed methods are implemented. The wrong way of changing the trait composition is shown in Figure 3.8

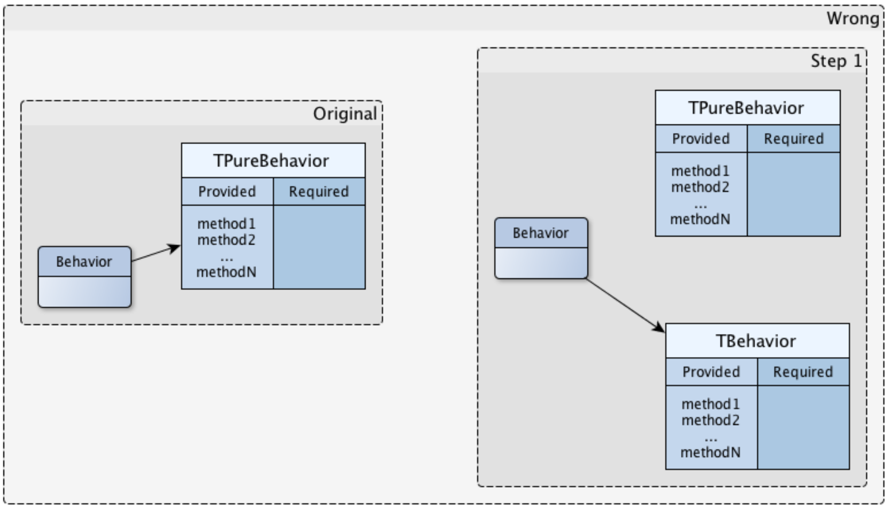


Fig. 3.8: Wrong way to make Behavior use TBehavior

For achieving this we first copied all the methods from TPureBehavior to `Behavior` (step 1). Then we made the class `Behavior` use the trait TBehavior and remove TPure-Behavior (step 2), and finally we removed all the methods from `Behavior` that we had copied (step 3).

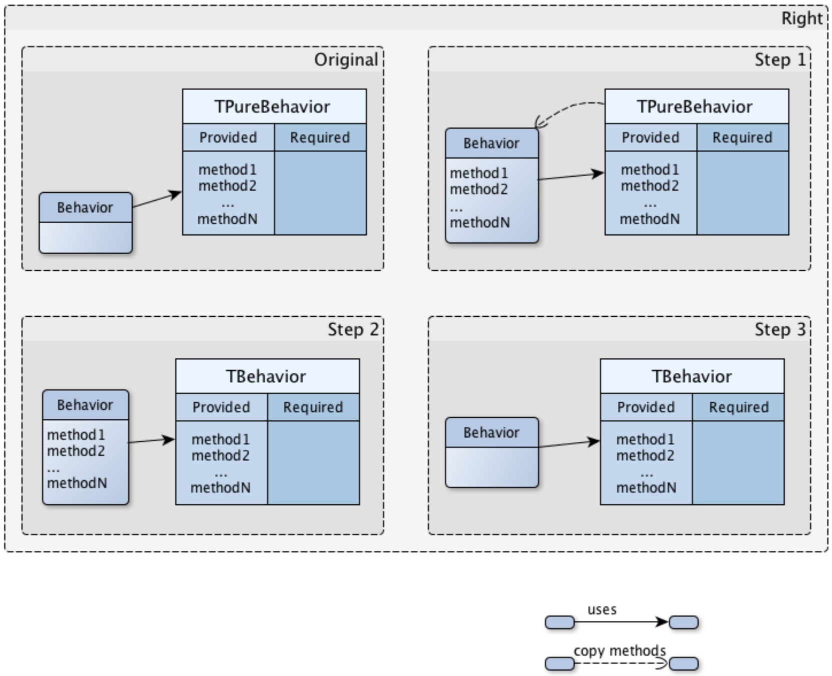


Fig. 3.9: Right way to make Behavior use TBehavior

The process was analogous for making `TraitBehavior` use `TBehavior`, `ClassDescription` and `TraitDescription` use `TClassDescription` and finally `Class` and `Trait` use `TClass`.

3.7. Integration challenge

Like the implementation, integration in Pharo was not an easy task. To begin, Monticello has problems when kernel classes are modified, this is so because when Monticello loads packages it needs to send messages to instances of kernel classes.

At the time of the integration, Monticello also had several troubles with handling traits. For example, when a method of a trait was modified, Monticello did not detect the change in the method and therefore it was not possible to commit.

The solution to these problems was replacing Monticello by `change sets`. However, `change sets` had problems as well. At the time of the integration appeared dependency problems, errors of methods no implemented but implemented later in the `change set`.

Something important was that between integration and integration, there should not have been red tests since the integration is not allowed when this happens. This was also a factor that hindered the integration, since sometimes some of the `change sets` needed to be rebuilt in order to not produce red tests.

Finally, change sets were divided in groups of common behaviors

- copying methods from the trait to the class
- copying methods from the class to the other trait

- changing the trait composition
- removing repeated methods
- re-categorizing methods
- fixing details such as adding or removing missing methods

The re-categorization of methods was needed because there was a bug in Pharo, that sometimes, when a `change set` was applied, the categorization was wrong made.

It is important to mention that some of the bugs presents in Monticello were fixed after the integration of these changes. However, Monticello is still not working correctly when kernel classes are modified.

In Figures 3.10, 3.11 and 3.12 you can see a summary of the classes after all the merge and movement of the methods.

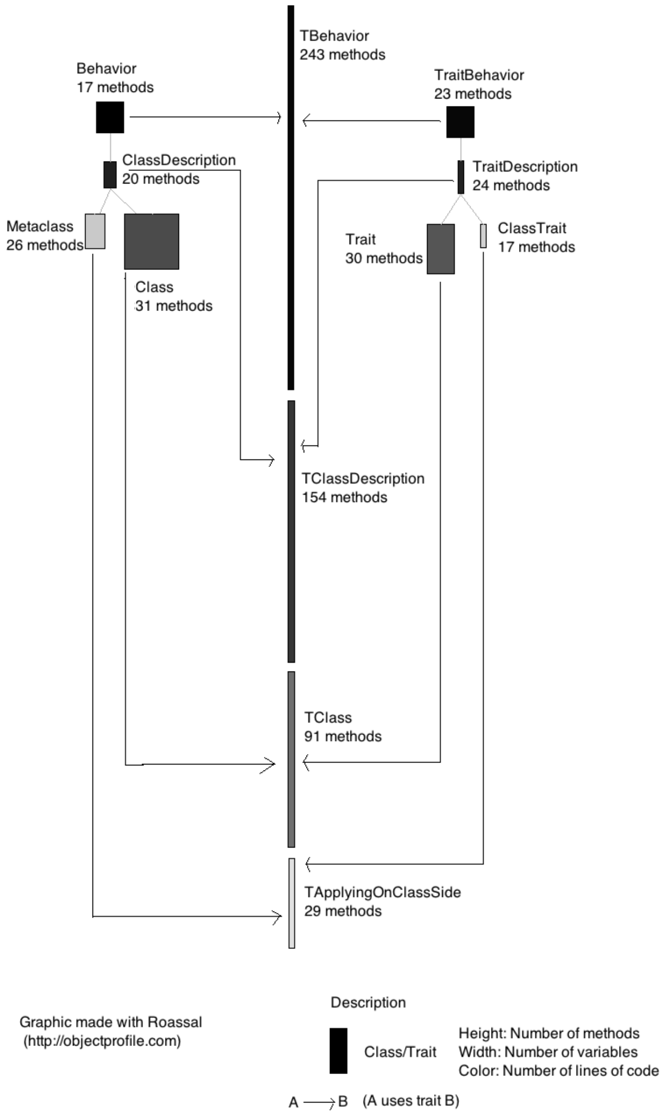


Fig. 3.10: Kernel classes and traits are now polymorphic

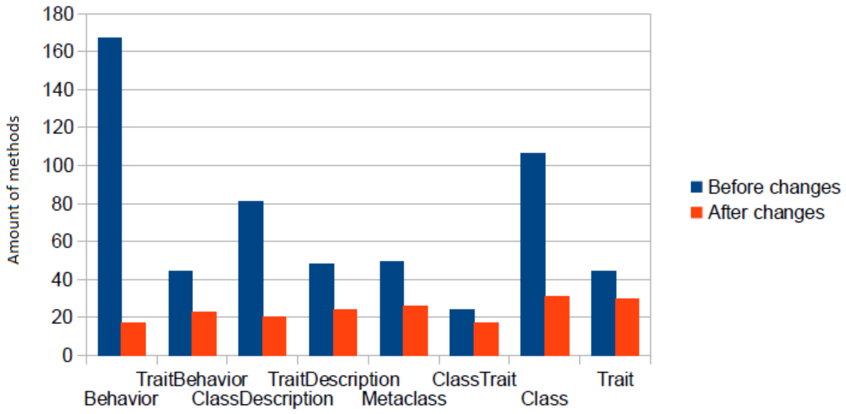


Fig. 3.11: Amount of methods in classes before and after changes

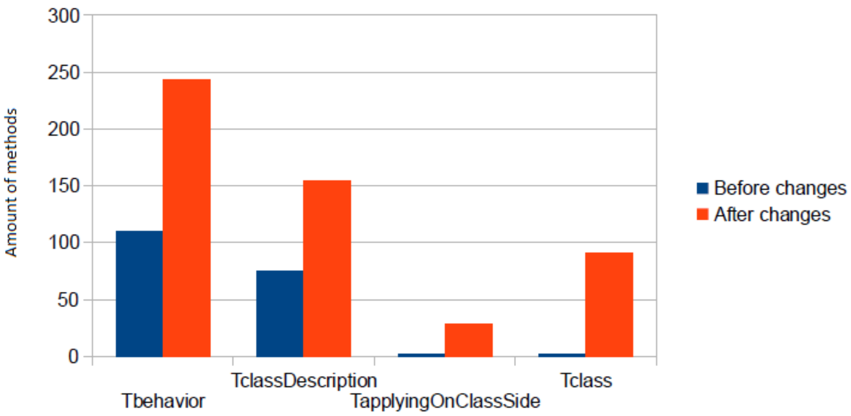


Fig. 3.12: Amount of methods in traits before and after changes

3.8. Advantages and disadvantages

One of the benefits of having polymorphism between classes and traits is that tools that know how to handle classes, do not need to change to handle traits too. Tools can send messages to traits in the same way that they do with classes. This is a great benefit for Pharo since programmers do not need to change tools with the inherent risk of introducing bugs.

Disadvantages of making them polymorphic is that the model becomes less consistent, why should a trait know how to answer messages like `superclass` or `instanceVariables`? Why show a class know how to answer `users`?

We decided that the benefits were more important and the integration was made.

3.9. Keeping polymorphism

After making classes and traits polymorphic, we had to make sure that they will remain so.

The way we did this was by adding unit tests that fail if the polymorphism is broken. There are four tests, one for `Behavior/TraitBehavior`, one for `ClassDescription/-TraitDescription`, one for `Class/Trait` and one for `Metaclass/TraitClass`.

These tests check 4 things (I will explain it for behavior but is analogous for the rest)

- If there is a repeated method in `Behavior` and `TraitBehavior`, it must access at least one of the instance variables, otherwise it can be implemented in `TBehavior`
- If the method is in `Behavior` and `TraitBehavior`, but with different implementations, it must be declared in `TBehavior` as an `explicitRequirement` method
- The methods in `TraitBehavior` minus the methods in `Behavior` is just a minimum set of allowed methods
- The methods in `Behavior` minus the methods in `TraitBehavior` is just a minimum set of allowed methods

This minimum set contains methods that are exceptions to the other two rules. We will give a description of those methods and why they are exceptions.

For `Behavior` minus `TraitBehavior` the set contains the methods {`format isBehavior layout`}

- `format` - The instance variable format belongs to `Behavior` but not to `TraitBehavior`. Traits do not know how to answer this message. It is an exception to polymorphism.
- `isBehavior` - The method returns true for all the classes but false for the traits, the implementation for traits (and the rest of the objects) is implemented in the `Object` class
- `layout` - Is an instance variable of `Behavior`, not of `TraitBehavior`. As we saw, this method is an exception to polymorphism. Traits do not know how to answer this message.

For `TraitBehavior` minus `Behavior` the set is {`localSelectors localSelectors: basicLocalSelectors basicLocalSelectors: browse isTrait`}

- `localSelectors, localSelectors: basicLocalSelectors, basicLocalSelectors:` The instance variable `localSelectors` belongs to `TraitBehavior` but not to `Behavior` - For classes these methods are implemented in `Class` and `Metaclass`
- `browse` - Is implemented differently for traits, the implementation for classes is in `Object`
- `isTrait` - Answers true for traits, for the rest of the objects answers false, the implementation is in `Object`

For `ClassDescription` minus `TraitDescription` the set is {`supeclass:layout: initializeLayout baseClass superclass:withLayoutType:slots: classClass initializeLayoutWithSlots layoutSized classVersion`}

- `superclass:layout, initializeLayout, initializeLayoutWithSlots:, layoutSized` - These methods are related with the instance variable `layout` which it has no sense for traits
- `baseClass classClass classVersion` - These methods have their equivalents in `TraitDescription` (`baseTrait classTrait traitVersion`). These methods are not polymorphic, the problem comes from the name of the selector

For `TraitDescription` minus `ClassDescription` the set is {`isClassTrait addExclusionOf: copyTraitExpresion baseTrait traitVersion isBaseTrait classTrait`}

- `isClassTrait, baseTrait, traitVersion, isBaseTrait, classTrait` - These methods have their equivalents in `ClassDescription` but with different name, as in the previous set, the problem comes from the name of the selector
- `copyTraitExpresion, addExclusionOf:` - These methods are not polymorphic with classes

For `Class` minus `Trait` the set is {`traitComposition: baseClass basicLocalSelectors clasClass localSelectors localSelectors: basicLocalSelectors: traitComposition addInstVarNamed:`}

- `traitComposition, traitComposition:, localSelectors, localSelectors:, basicLocalSelectors, basicLocalSelectors:` - For traits the implementation is in `TraitBehavior`
- `baseClass classClass` - These methods have their equivalent one in `Trait`, but the problem is again, the name of the selector
- `addInstVarNamed:` - For traits it is implemented in `TraitDescription`

For `Trait` minus `Class` the set is {`classTrait isClassTrait classTrait nautilusIcon isBaseTrait initialize baseTrait`}

- `classTrait isClassTrait classTrait: isBaseTrait baseTrait` - These methods have their equivalent one for classes with different name
- `nautilusIcon initialize` - These methods are implemented differently for traits, classes have their implementation in `Object`

For `Metaclass` minus `ClassTrait` the set is {`baseClass basicLocalSelectors basicLocalSelectors: classClass environmentfileOutOn:moveSource:toFile:. fileOutOn:moveSource:toFile:initializing: localSelectors localSelectors: postCopy traitComposition traitComposition: veryDeepCopyWith:`}

- `basicLocalSelectors basicLocalSelectors: localSelectors localSelectors: traitComposition traitComposition:` - The implementation for classTraits is in `TraitBehavior`

- `baseClass classClass` - These methods have their equivalent one for `ClassTraits` (`baseTrait classTrait`)
- `environment postCopy` - The implementation for classTraits is in `TBehavior`
- `veryDeepCopyWith:` The implementation for classTraits is in `Object`
- `fileOutOn:moveSource:toFile: fileOutOn:moveSource:toFile:initializing:` - The implementation for classTraits is in `TClassDescription`

For `ClassTrait` minus `Metaclass` the set is {`initializeWithBaseTrait: asMCDefinition baseTrait: isClassTrait baseTrait compile:classified:withStamp:notifying:logSource: isBaseTrait copy classTrait:`}

- `initializeWithBaseTrait: asMCDefinition` - These methods have no equivalent in metaclasses, therefore these messages are not polymorphic
- `baseTrait isClassTrait classTrait baseTrait classTrait:` - These methods have their equivalent one for metaclasses
- `compile:classified:withStamp:notifying:logSource:` - The implementation for Metaclasses is in `TClassDescription`
- `copy` - The implementation for metaclasses is in `Object`

3.10. Conclusions

When you are working on a live objects environment, where the changes affect the development environment, modifying kernel classes can be complicated.

Integrations were made step by step generating several `change sets`. This caused that the time since we started with the implementation to the moment that was everything integrated was long.

About making classes and traits answer messages that should not, is quite questionable. It is not good design put responsibilities to classes and traits that do not belong to them, but as we saw before, it brings benefits.

It is difficult to know if the polymorphism fulfilled its purpose. To know this we would have to modify tools to not distinguish traits from classes. We had not time to do this and therefore is difficult to draw conclusions regarding to the changes made.

What we did as you will see in the next chapter, is to import the implementation of refactorings made for Squeak by Alejandro Gonzalez. This was done for two reasons. One because refactorings of traits were necessary in Pharo since there was none. Another reason was for checking that the system continue working properly after the changes.

Since the day that the integration was made to the present day, there were almost no bugs reported related to the changes made. Everything worked properly.

4. REFACTORINGS FOR TRAITS

4.1. Introduction

After making classes and traits polymorphic, it was needed to check that everything kept working properly. We decided to start using the system in order to test it.

The usage of the system was made by the RMoD team (the team in charge of pushing Pharo) and for us. People from RMoD used it for their own projects and we used it for implementing refactorings for traits. Refactorings for traits did not exist in Pharo, so we thought it was a good opportunity to implement them. Doing this we "kill two birds with one stone", we added refactorings for traits and we tested the system with the changes made for the polymorphism.

There were already refactorings for traits in Squeak, they were made by Alejandro Gonzalez in his master thesis, thus, we decided to import them to Pharo instead of doing a new implementation.

4.2. Importance of refactorings

The refactorings of code are a very useful tool during development, they help to modify code keeping the behavior of the system.

The goal of refactorings is to improve the ease of understanding of code or change its structure and design and eliminate dead code, to facilitate future maintenance.

Pharo did not have any kind of refactorings for traits, this was one of the reasons why we decided to import them. While traits are not currently widely used, providing tools to facilitate their handling can change this.

4.3. Importance of testing the system

As we saw in Chapter 3, modify the kernel classes of the system were required to achieve polymorphism. This can cause serious problems in a development environment such as Pharo, which is programmed using those same classes. Testing the system was absolutely necessary to avoid possible future problems.

4.4. Refactorings added

Let's see now the refactorings added to Pharo

- Create missing requirements
- Move method to trait
- Extract method to trait
- Extract trait

- Flatten trait
- Use trait

4.4.1. Create missing requirements

This refactoring consists of making all the implicit requirements in a trait explicit. The requirements in a trait are all those methods that are not provided by itself, this is all the messages sent to the pseudo-variable `self` that are not implemented in the trait.

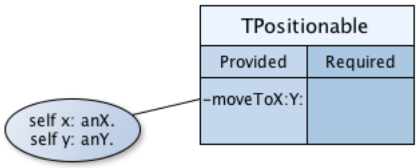


Fig. 4.1: Before executing the refactoring

Figure 4.1 shows that the method `moveToX:Y:` is sending the messages `x:` and `y:` to `self`, but `x:` and `y:` are not implemented in the trait (not even as a requirements), these two methods should be explicit requirements since if the user does not implement them, then it won't be able to answer the message `moveToX:Y:`.

After executing the refactoring the methods are added, as you can see in Figure 4.2

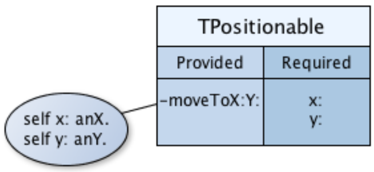


Fig. 4.2: After executing the refactoring

4.4.2. Move method to trait

This refactoring consists in moving a method in a class or trait, to one of the traits in its trait composition. If the method to be moved is using instance variables, the refactoring creates accessors to those instance variables in the class and make them requirements in the trait.

As you can see in Figure 4.3, the class `Rectangle` is using the trait TPositionable, `Rectangle` has 2 instance variables which indicate a point in 2-dimentional space (`x` and `y`), if we move the method `moveTenPositionsLeft` to the trait `TPositionable` (using the refactoring), we will notice how the accessors to the variables are created in `Rectangle` Figure 4.4, also, in the implementation of `moveTenPositionsLeft` the variables are replaced by the accessors methods and are declared as requirements in the trait.

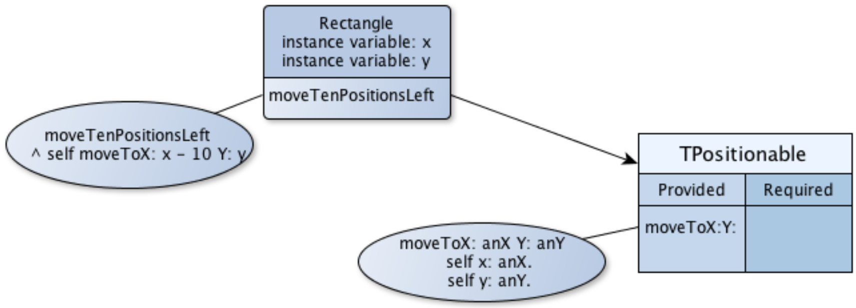


Fig. 4.3: Before executing the refactoring

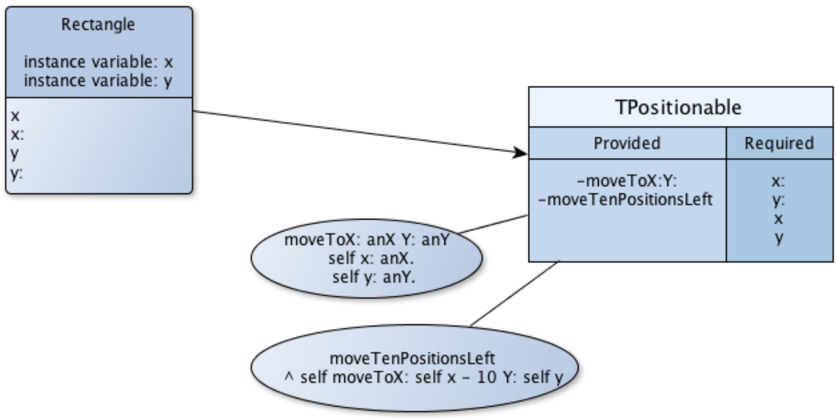


Fig. 4.4: After executing the refactoring

4.4.3. Extract method to trait

This refactoring takes a fragment of code of a method, and with it creates a new method in some selected trait. For this purpose, the refactoring extracts first the selected fragment of code by using the `Extract method` refactoring, and then it moves the extracted method to the trait using the refactoring `Move method to trait`

4.4.4. Extract trait

This refactoring takes a set of methods in a class (or trait) and creates a new trait from them. If the trait already exists, the refactoring completes the trait with the selected methods.

4.4.5. Flatten trait

This refactoring flattens in a class or trait, all the methods that a determined trait provides to it, this is achieved by compiling all the methods of the trait in the user class (or trait). Those methods that are implemented both in the user and the trait, are not compiled. By doing this, all the methods are then defined locally. Exclusions are not flattened and aliases are flattened using the alias as a selector.

In Figure 4.5 the class `C1` is using the trait `T1`, it creates an alias called "method" for `m3` and excludes `m3`, doing this, an instance of `C1` knows how to answer the messages `m1`, `m2` and `m3`

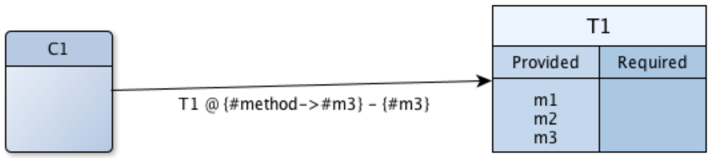


Fig. 4.5: Before executing the refactoring

After executing the refactoring (by flattening `T1` into `C1`) Figure 4.6, all the methods are compiled in `C1`, the method `m3` is compiled as `"method"` (because of the alias) and the composition is removed from `C1` (`C1` does not use `T1` anymore).

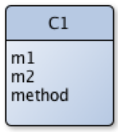


Fig. 4.6: After executing the refactoring

4.4.6. Use trait

This refactoring adds a trait to the trait composition of a class, removing from the class all those methods duplicated with the trait recently included. The methods considered to be removed are those whose definition is the same in the trait.

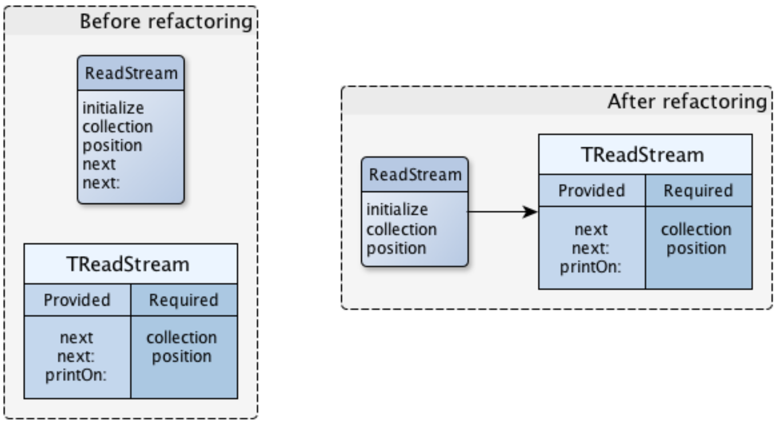


Fig. 4.7: Use trait refactoring

As you can see, after refactoring, the repeated methods (in this case `next` and `next:`) are removed from the class. We are not showing the implementation of the methods but it is supposed to be the same in the class and the trait.

4.5. Migrating to Pharo

As said previously, the implementation is based on the one made for Squeak by Alejandro Gonzalez in his master thesis [10]. The refactoring classes were extended to support traits.

The import process was not complicated but it took time. The needed classes were imported using the fileIn/Out tool and then all the needed changes were made by hand. Some of these changes were renaming of methods, adding comments, testing classes, etc.

Most of the test cases were imported, we needed to rename some of them to keep some kind of consistency with the existing tests in Pharo. After all the importing, the test cases were working except for a few of them that failed because a bug (not related with traits) in the package organizer of Pharo. Due this bug, the refactorings were not integrated yet in Pharo, but the idea is to have them for the stable version of Pharo 3.

4.6. Conclusions

As mentioned in the introduction, we had two goals, testing the system and provide Pharo of refactorings for traits.

Regarding to the testing of the system, we can say that there were not complications. During the importing process all the system worked quite good. We had some problems with methods that were improperly integrated in the image. We can mention for example the method `isBehavior` that answers true for classes and false for traits. During the integration this methods was wrongly copied to the shared trait `TBehavior` and therefore traits answered true to this message causing a large number of red tests.

From the RMoD team side, there were some suggestions.

- Find a name to represent both a class and a trait: Some of the developers are using the word `behavior` but it is not well accepted by others.
- Some of the developers disagree about what `isBehavior` must return for traits: Some developers say that it must return `true` since they provide a behavior, but others say that a behavior is an object that can create instances.
- Re-think some the methods names to refer the hierarchy of classes and traits (theMetaclass, theNonMetaclass, classSide, classTrait, etc): This point is related to the previous one, it is needed to define names and what they should return for referencing the class hierarchy. As an example we can think of the method `classTrait`, traits answer the “metaclass” of the trait, but classes do not know how to answer this message.

Regarding to the refactorings, unfortunately they were not integrated yet because problems with packages that cause red tests. We cannot say if they were useful to the community but we can say that they are working and can be also integrated in the stable version of Pharo 3 (after the bug of the package organizer is fixed).

As a summary, the image worked quite good and there were no big problems reported. We can say that polymorphism changes can be used in the stable version of Pharo 3.

5. EXPLICIT REQUIREMENT METHODS BUG

5.1. Introduction

Traits can have requirements. Requirements are methods that the user have to implement. Traits provide methods that can depend on required ones.

For example in Figure 5.1 we see that `TMovable` has two required methods (`x:` and `y:`) to set the coordinates of the user. In the implementation of `x:` and `y:` (in the trait) we have `^self explicitRequirement`, which means that the user of the trait has to implement the method. If the user (in this case Rectangle) does not do it, when either the message `x:` or `y:` is sent to an instance of Rectangle, the program will throw an exception.

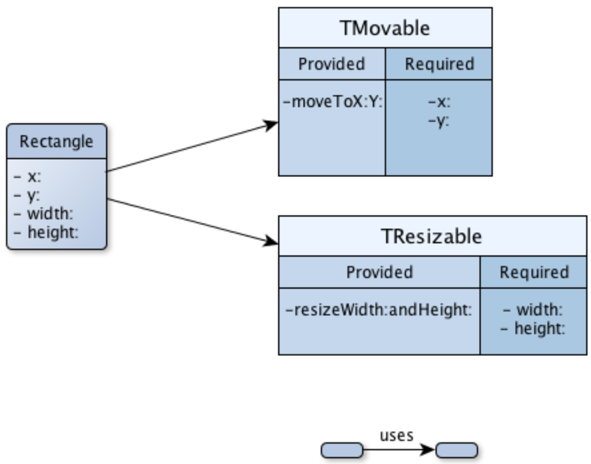


Fig. 5.1: Class `Rectangle` using traits `TMovable` and `TResizable`

In the current implementation of Pharo, there is a bug in explicit requirement methods. For example, if we have a hierarchy like Figure 5.2, when the message `m` is sent to an instance of `C2`, an error message is thrown saying that the method is explicit requirement, despite of the fact that `m` is implemented in the superclass.

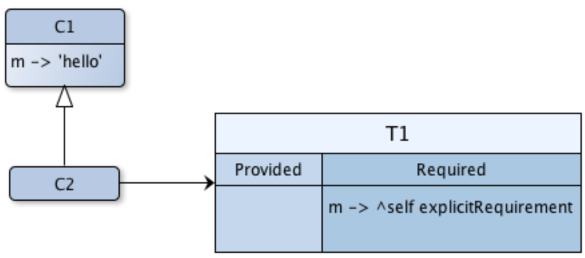


Fig. 5.2: Class `C2` with superclass `C1` and using trait `T1`

This problem occurs because when a trait is added to a class, all the methods of the trait are merged together with the methods of the class. Internally, the method dictionary of the user will also contains the methods of the trait.

5.2. Possible solutions and their problems

1. Modify the method lookup to check if the method is an explicit requirement
2. Update the method dictionary accordingly to the case
3. Modify the implementation of `Object>>explicitRequirement`

The idea of 1 is to modify the method lookup to check if the method found is an explicit requirement method. If the method is a requirement then it is ignored and the VM continues with the superclasses, if the method is not found in the superclasses then the explicit requirement error is thrown. The problem with this solution is that it requires to modify the Pharo virtual machine, and this is not an easy task. Besides, traits do not exist at the VM level.

In 2, the idea is to update the method dictionary accordingly (remember that internally, classes and traits store the methods of its trait composition in the method dictionary). The problem with this solution is that there are many cases to consider.

What happens if in Figure 5.3 you add the method `m` to `C1`?

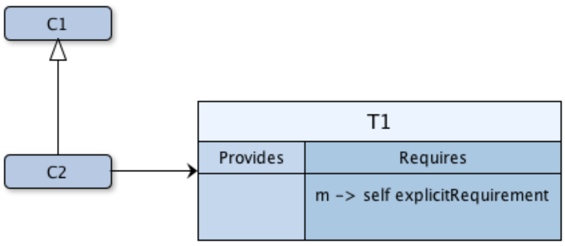


Fig. 5.3: Class `C2` with superclass `C1` and using trait `T1`

In this case we need to see if the new method is an explicit requirement method. If so, we have to check if one of the superclasses of `C1` can perform `m`, and in this case, do not add it to the class dictionary.

If the method is not an explicit requirement method, we have to check in all the subclasses of `C1` if any of them has the method as an explicit requirement. In those ones that have it (in this case `C2`), we remove `m` from their method dictionary in order to give priority to `C1>>m`.

Let's see now a different case.

What happens if we add the method `m` (explicit requirement) to `T1` in Figure 5.4?

In this case we need to check if the method is an explicit requirement (in this case it is) and if one of the superclasses knows how to perform it (in this case `C1` knows), then we do not add the method to `C2` dictionary.

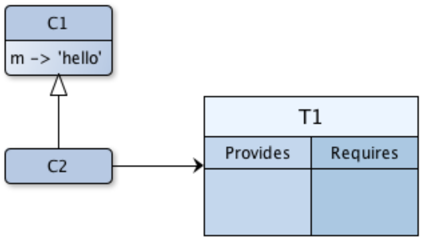


Fig. 5.4: Class `C2` with superclass `C1` and using trait `T1`

The problem with this solution is that while it works, it is not an elegant solution, modifying the dictionary of methods needs modification of key methods (when adding methods, when removing methods, etc), and might be cases in which it will not work.

The final solution, and the one that was integrated in Pharo, was to modify the implementation of `Object>>explicitRequirement` such that this method is responsible for looking for the right implementation in the class hierarchy.

Consider the hierarchy in Figure 5.5

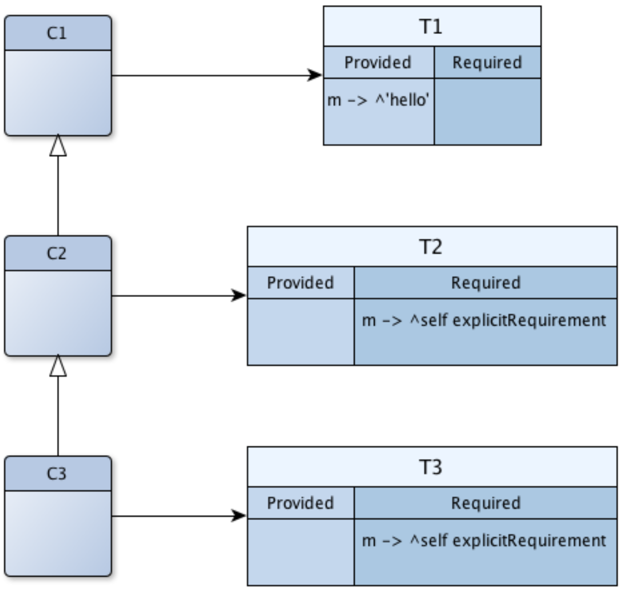


Fig. 5.5: More complex hierarchy of classes and traits

If you send the message `m` to an instance of `C3`, then in the implementation of `explicitRequirement` goes through the class hierarchy of `C3` looking for a class that can perform `m`, if it finds it, it executes the method, otherwise it throws the explicit requirement method error.

Below is the final implementation

```
explicitRequirement
  | originalMethod originalSelector originalClass originalArguments errorBlock |
```

```
1 -  originalClass := self class.
2 -  originalMethod := thisContext sender method.
3 -  originalSelector := originalMethod selector.
4 -  originalArguments := thisContext sender arguments.
5 -  errorBlock := [ ^ self error: 'Explicitly required method' ].
6 -  originalMethod isFromTrait
7 -    ifFalse: errorBlock.
8 -  originalClass superclass
9 -    withAllSuperclassesDo: [ :superCl |
10 -       superCl methodDict
11 -         at: originalSelector
12 -         ifPresent: [ :method |
13 -           (method isProvided or: [ method isFromTrait not ])
14 -             ifTrue: [^method valueWithReceiver: self arguments: originalArguments]]
15 -         ifAbsent: [ ]].
16 -  ^ errorBlock value
```

- In lines 1, 2, 3 and 4 we obtain information from the context such as: senders, receivers, method sent, arguments, etc.
- Line 5 declares the block that will be thrown in case of not finding the method
- Line 6 and 7 check if the method executed is coming from a trait, if this is not the case it throws the error. This is so because someone can declare the explicit requirement in the class on purpose
- Line 8 and 9 start the cycle from the superclass of the receiver in order to look for the implementation of the method.
- Line 10, 11, 12 and 13 check if the method is in one of the superclasses and if the method is provided (this means that it is not another explicit method)
- Line 14 means that the method was found in one of the superclasses, and then it explicitly executes that method for the receiver with the corresponding arguments
- Finally, line 16 means that the method was not found so it throws the error message.

The next sequence diagram is a simplification of what happens when the message `m` is sent to an instance of `C3`.

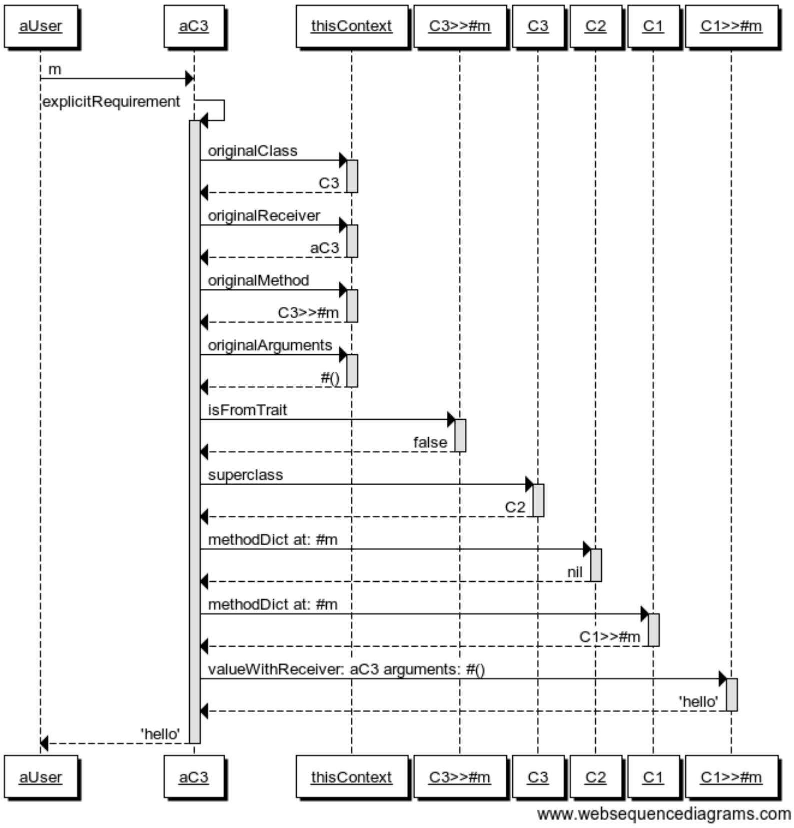


Fig. 5.6: Messages sent in case of finding an implementation of explicitRequirement

First, it is needed to know which is the object that received the message, the class of the object, the message that was sent and the parameters. Then, it is needed to check if one of the superclasses can perform that message. As you see in Figure 5.6 the class `C2` is checked first (the superclass of `C3`), `C2` does not know how to perform the method so `C1` is checked later (the superclass of `C2`). `C1` knows how to perform the method so it is executed with the corresponding parameters.

Now, let's suppose that the method `m` is not implemented in the trait `T1` Figure 5.5, then `C1` does not have the method `m` in its method dictionary, therefore it executes an error block.

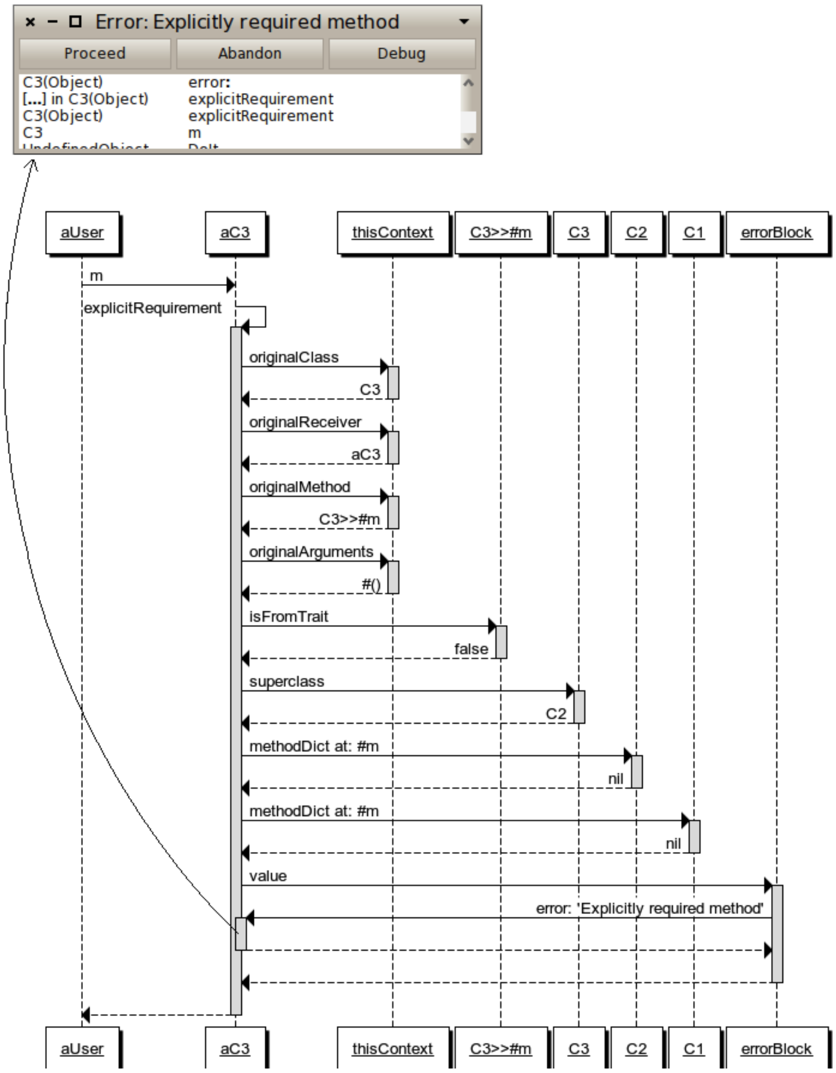


Fig. 5.7: Messages sent in case of not finding an implementation of explicitRequirement

One of the problems with this solution is that you have to add the return symbol in the method

```
m -> self explicitRequirement (WRONG)
m -> ^self explicitRequirement (RIGHT)
```

This is needed because if it finds a method in the superclass that effectively can perform the selector, this method could return a value, and if this is the case the explicitRequirement method will not be able to do it.

Another problem is that this solution can be slow since the lookup is made every time dynamically (and not through the VM), this can be solved by excluding the explicit required methods from the trait composition. for example in Figure 5.5 the solution to be more performant would be to exclude `m` from `C3` trait composition and exclude `m` from `C2` trait composition.

```
C1 subclass: #C2
   uses: T2 - {#m}
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MyCategory'
```

```
C2 subclass: #C3
   uses: T3 - {#m}
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MyCategory'
```

5.3. Testing

Since we have different cases to consider, it is important to keep control of modifications during the implementation process. Therefore, the implementation was made using TDD [14]

These are the cases taken into account:

1. `testExplicitRequirementDoesNotTakePrecedenceEvenWhenAddingTraits`
2. `testExplicitRequirementDoesNotTakePrecedenceInDeepHierarchy`
3. `testExplicitRequirementInClassAlwaysTakesPrecedence`
4. `testExplicitRequirementTakesPrecedenceOverTraitImplementation`
5. `testExplicitRequirementWithSuperclassImplementationAndAnotherTrait`

Now, let's see them more in detail

Case 1: `testExplicitRequirementDoesNotTakePrecedenceEvenWhenAddingTraits`

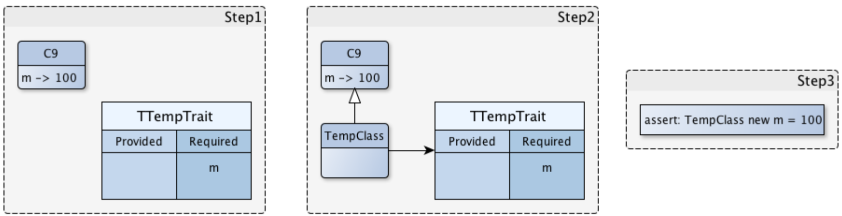


Fig. 5.8: Test Case 1

Step 1:
The class `C9` is created with the method `m` that returns 100.
The trait `TTempTrait` is created with a required method `m`

Step 2:
A new subclass of `C9` called `TTempClass` is created. It uses the trait `TTempTrait`.

Step 3:
Instances of `TTempClass` should return 100

Case 2: `testExplicitRequirementDoesNotTakePrecedenceInDeepHierarchy`

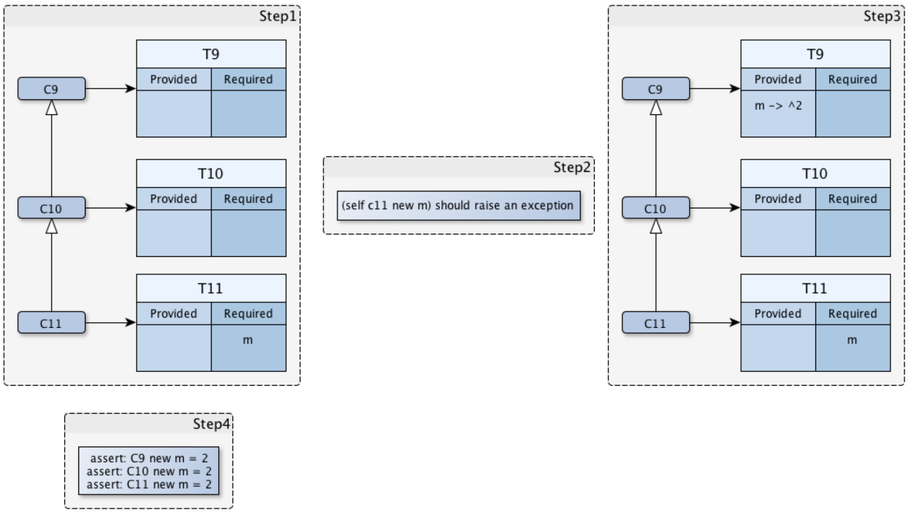


Fig. 5.9: Test Case 2

Step 1:
The hierarchy of classes is created.

Step 2:
Instances of `C11` should raise an exception when they receive the message `m`

Step 3:
A method `m` that returns 2 is compiled in `T9` (the trait used by the superclass).

Step 4:
Now, instances of `C11`, `C10` and `C9` should return 2 when they receive the message `m`

Case 3: `testExplicitRequirementInClassAlwaysTakesPrecedence`

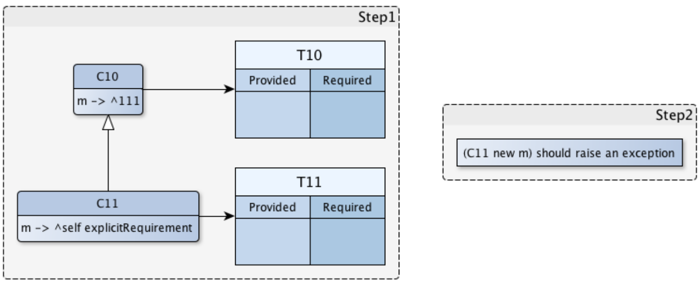


Fig. 5.10: Test Case 3

Step 1:
The hierarchy of classes is created.

Step 2:
Instances of `C11` should raise an exception when they receive the message `m`. This is so because when the explicit method is compiled in the class, we assume that the programmer

did it on purpose and want the error to be thrown.

Case 4: `testExplicitRequirementTakesPrecedenceOverTraitImplementation`

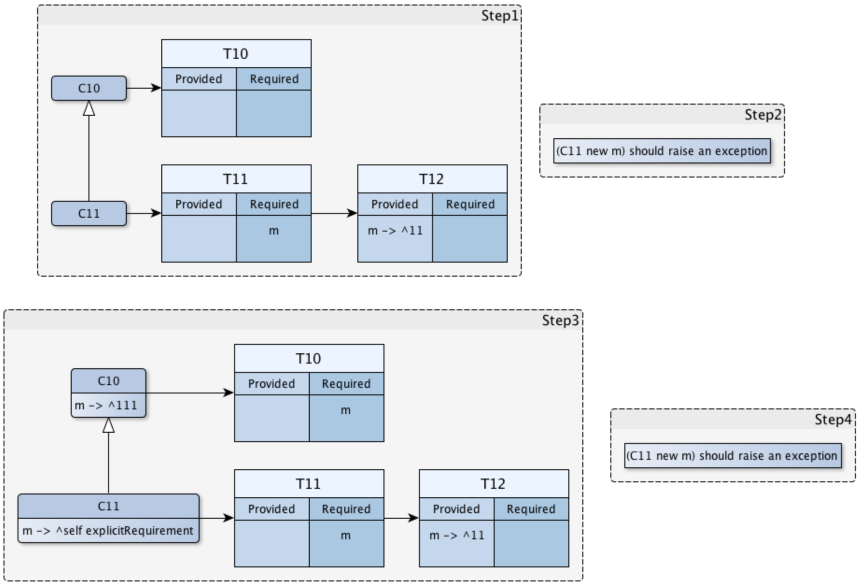


Fig. 5.11: Test Case 4

Step 1:
The hierarchy of classes is created.

Step 2:
Instances of `C11` should raise an exception when they receive the message `m`.

Step 3:
A method `m` that returns 100 is compiled in `C10`. An explicit required method `m` is compiled in the trait `T10`.

Step 4:
Instances of `C11` should still raise exceptions when they receive the message `m`. This is so because the explicit requirement method `m` is implemented in the class.

Case 5: `testExplicitRequirementWithSuperclassImplementationAndAnotherTrait`

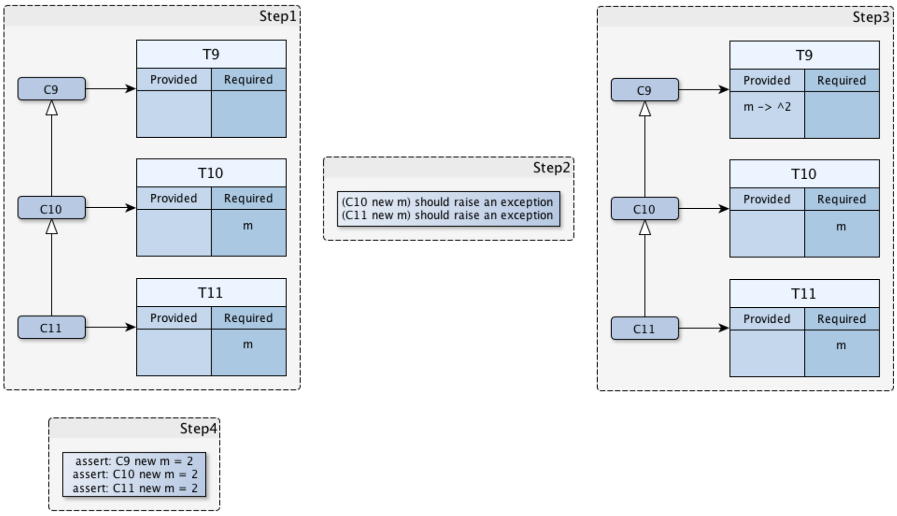


Fig. 5.12: Test Case 5

Step 1:
The hierarchy of classes is created.

Step 2:
Instances of `C10` and `C11` should raise an exception when they receive the message `m`.

Step 3:
A method `m` that returns 2 is compiled in `C9`.

Step 4:
Instances of `C9`, `C10` and `C11` should return 2 since the implementation is in the trait of the superclass.

5.4. Conclusions

This bug was not trivial to fix. As we saw, the immediate solution was to modify the method dictionary to keep it updated according to the changes we are making.

The solution generated distrust in some of the developers at the RMoD team for two reasons. First, it was needed to modify some of the compiling methods. These are sensitive methods and modify them can bring unknown problems. Second, there are different cases to take into account, it is possible to have cases that we did not think about. As we saw, we decided to look for a different solution.

Modifying the implementation of `explicitRequirement` was more accepted by the team, is more elegant and was accepted. All the cases we saw before worked properly and this solution was integrated in Pharo.

6. TRAIT COMPOSITIONS AND THE PARENTHESIS PROBLEM

6.1. Introduction

Traits can be composed to form a trait composition. Then a class (or other trait) can use this trait composition and have all its methods. See Figure 6.1 (methods with `T` means that come from a trait). `C1` is using the composition `T1 + T2`, and `T1` is using the composition formed by the single trait `T3`.

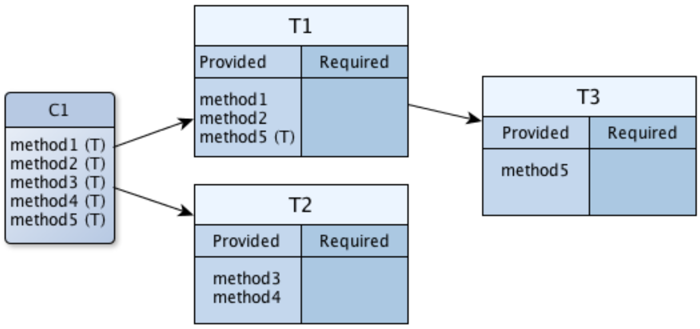


Fig. 6.1: Example of a simple trait composition

A trait composition has the flattening property, which says that the semantics of a method defined in a trait is identical to the semantics of the same method defined in a class that uses the trait. Figure 6.2

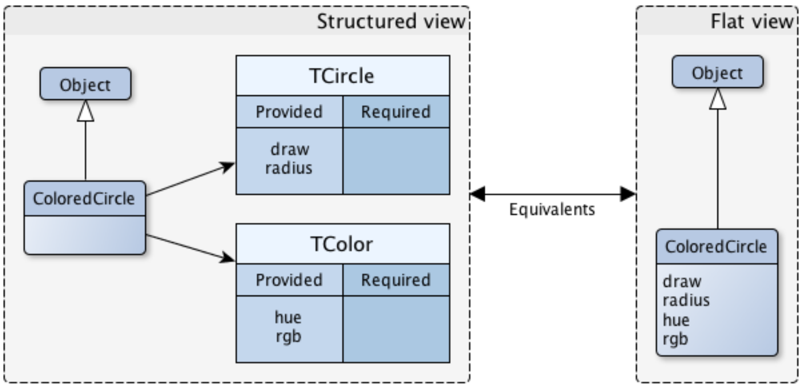


Fig. 6.2: Structured View vs Flattened View

Also, the composition order is irrelevant, and hence conflicting trait methods must be explicitly disambiguated. Figure 6.3

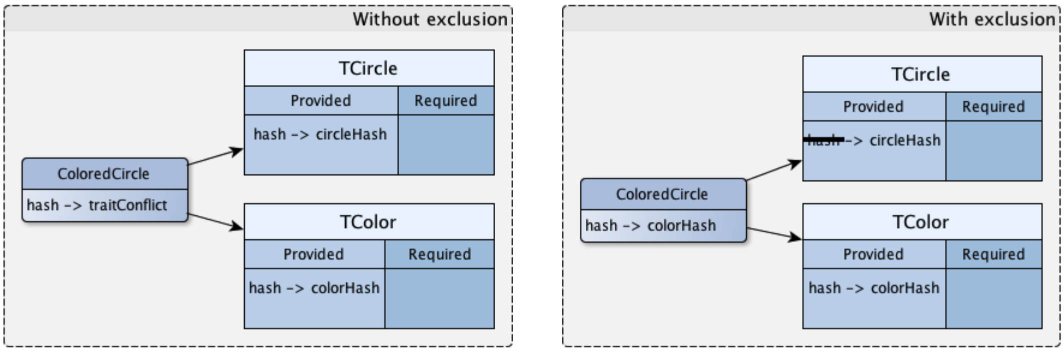


Fig. 6.3: Example of trait composition with and without exclusions

Trait compositions are formed by transformations, a transformation can be either a trait, an exclusion or an alias. The form of a composition is like this: `T1 + T2 + ... + Tn` where `Ti` is a transformation.

Transformations can be composed to form other transformations, but taking into account some restrictions. Exclusions have to be applied after aliases, a composition like `T1 - {#m1} @ {#m2->#m1}` throws an exception. The correct composition would be `T1 @ {#m2->#m1} - {#m1}` .

Another restriction to take into account is that exclusions cannot be applied after another exclusion. The composition `T1 - {#m1} - {#m2}` throws an exception too. The right composition would be `T1 - {#m1. #m2}`

As an example you can see in Figure 6.1 that `C1` is using the composition formed by 2 transformations: `T1 + T2`.

In Figure 6.3, before the application of the exclusion, the composition is `TCircle + TColor` and after the application of the exclusion, `ColoredCircle` uses the composition: `TCircle - {#hash} + TColor`.

6.2. Conflicts resolution

In Figure 6.4 you can see that there is a conflict in `C1`. This is because `C1` is using two traits with the method `m1`, but with different implementations. When the message `m1` is sent to an instance of `C1`, an error is thrown. To solve this problem, we can either exclude or alias one of the methods.

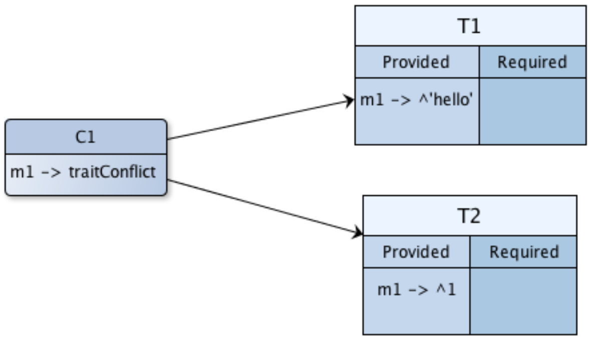


Fig. 6.4: Method with the same name in two different traits

For example, C1 can be defined as follows

Object subclass: #C1
 uses: T1 + T2 - {#m1}
 instanceVariableNames: ''
 classVariableNames: ''
 poolDictionaries: ''
 category: 'MyCategory'

Here, we are saying to the class, use the traits `T1` and `T2`, but ignore the method `m1` from `T2`. We can see the result in Figure 6.5

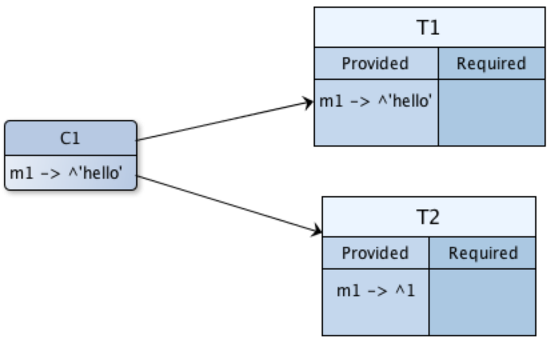


Fig. 6.5: Excluding `m1` from `T2`

It is also possible to create an alias for a method. Therefore we can define `C1` like this

Object subclass: #C1
 uses: T1 + T2 @ {#m2->#m1} - {#m1}
 instanceVariableNames: ''
 classVariableNames: ''
 poolDictionaries: ''
 category: 'MyCategory'

This means that `C1` is using an alias of `m1` called `m2`, and ignores the method `m1` from `T2`. You can see the result in Figure 6.6.

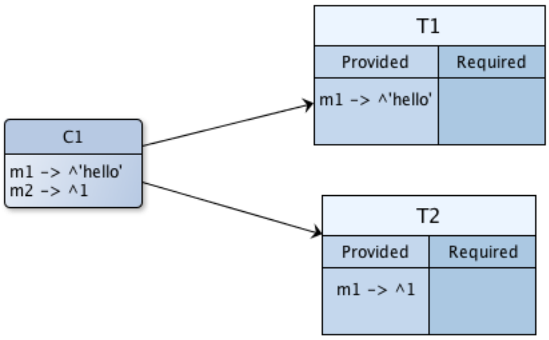


Fig. 6.6: Aliasing `T2>>m1` to `T2>>m2` and excluding `T2>>m1`

6.3. Problem

In the current implementation of Pharo (and Squeak), there is a bug when excluding methods. For example it is not possible to define a composition like `(T1 + T2) - #m`. And the same problem occurs for aliases, it is not possible to define the composition `(T1 + T2) @ (#m2->#m1)`

This problem occurs because the current implementation applies the exclusion/alias only to the last transformation inside the parenthesis. Therefore `(T1 + T2) - #m` would be interpreted as `T1 + (T2 - #m)`, and `(T1 + T2) @ (#m2->#m1)` would be interpreted as `T1 + (T2 @ (#m2->#m1))`. Clearly this is wrong and it needs to be fixed.

6.4. Fixing the problem

The problem was solved by applying exclusions and aliases accordingly to parenthesis. For example, `(T1 + T2) - #m1` now is equivalent to `(T1 - #m1) + (T2 - #m1)` which in turn is equal to `T1 - #m1 + (T2 - #m1)`

This change has some implications. Consider the composition `(T1 - {#a} + T2) @ {(#z ->#c)}`, in the old implementation the alias would be applied only to the last element, therefore, the composition is interpreted as `(T1 - {#a}) + (T2 @ {(#z ->#c)})` which has no problems. After the bug was fixed, the composition is interpreted as `(T1 - {#a} @ {(#z->#c)}) + (T2 @ {(#z->#c)})`, and this has a problem since, as we saw before, exclusions have to be applied after aliases.

Due this change, some of the compositions present in the system had to be modified. Let's see an example:

```
testPrinting
  | composition |
  composition := (self t1 - {#a} + self t2) @ { (#z -> #c) } - { #b. #c }.
  self assertPrints: composition printString
    like: 'T1 - {#a} + T2 @ {#z->#c} - {#b. #c}.
```

This test case does not work after fixing the bug, this is so because in the old implementation, the alias of the composition is applied only to `self t2`. After fixing the bug, the alias will be applied to both transformations. It is needed to change all the compositions in the system that have this same problem.

```
testPrinting
  | composition |
  composition := self t1 - { #a } + (self t2 @ { (#z -> #c) } - { #b. #c })
  self assertPrints: composition printString
    like: 'T1 - {#a} + (T2 @ {#z->#c} - {#b. #c}).
```

As you can see, not only the compositions was needed to change, but also the result of the composition (the printing), this required changes were not difficult but in a system with a large number can be quite tedious, specially in more complex compositions.

6.5. Conclusions

As you saw, one of the problems this modification has is that the semantic of the composition change. After we fix this bug, it was needed to modify some of the tests cases due exclusions applied before aliases. As a consequence, current implementations in Pharo will need to change if they want to have a correct semantic.

In systems with complex compositions, or with a large number of compositions, it will be needed to check each composition and modify them for the equivalent one.

7. FUTURE WORK

7.1. Hierarchy of classes and traits

As you saw, one of the classes to model traits is `TraitDescription`, the problem with this class is that its responsibility it is not well defined. `TraitDescription` was created to have a parallel hierarchy with classes, in which the equivalent class is `ClassDescription`.

During the development of this work, the idea of removing these classes was proposed by a member of the community. Instead of having these classes, the methods can be moved to the shared traits, therefore, all the methods from `TraitDescription` could be moved to `TBehavior` and be merged with the methods of `ClassDescription`. You can see the resulting hierarchy in Figure 7.1

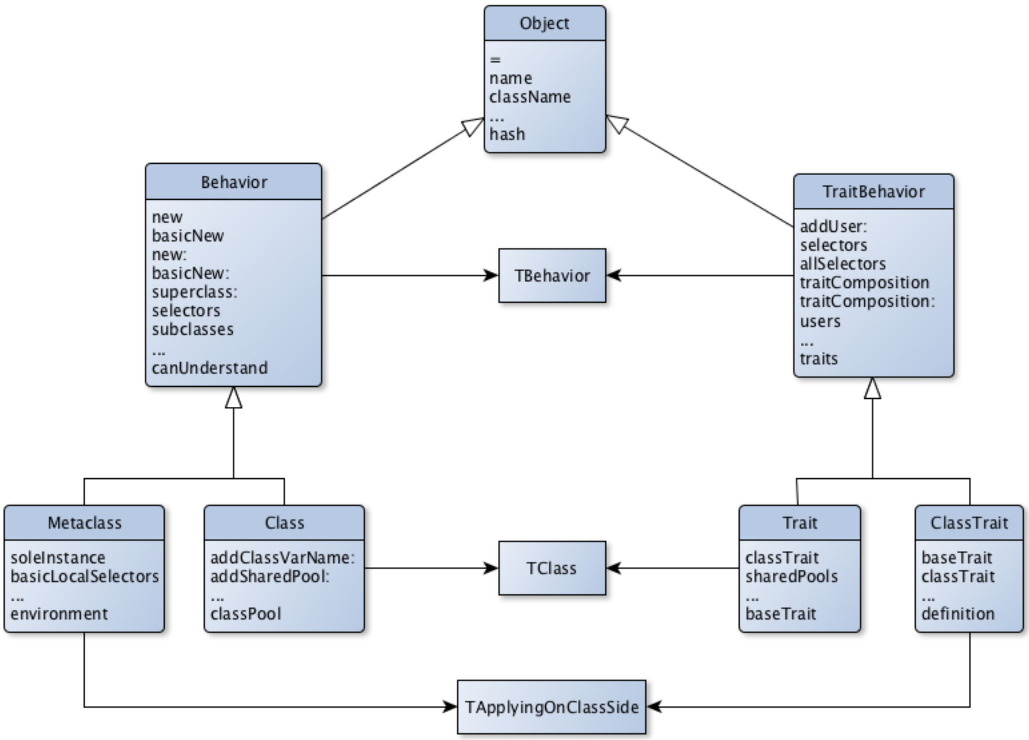


Fig. 7.1: Resulting hierarchy of removing ClassDescription and TraitDescription

The benefits of this would be to have a cleaner hierarchy of classes and traits.

7.2. Browser for traits

7.2.1. Introduction

From the beginning, Smalltalk systems were built around a tools metaphor, providing numerous tools for editing and manipulating the objects of the system. The basic tools included in a Smalltalk environment are: `Inspectors` for manipulating instances -Objects- `Browsers` to manipulate the classes, methods, and packages of the system, Workspaces for sending messages by evaluating expressions, and the `Transcript` as a global console for the system . Modern Smalltalk systems evolved to include `Test Runner` tools for running tests and refactoring browsers. [15]

When traits were integrated in Smalltalk, they were included as if they were classes, they appear in the second column of the browser and it is possible to see classes and traits equally, then, selecting a trait is possible to see its categories and its methods, just like classes.

7.2.2. Problem

One of the main problems in Pharo when dealing with traits is how to visualize them, currently the only way to do this is through Nautilus, the browser of classes.

Nautilus shows classes and traits in the same manner, and when a class or trait has a trait in its trait composition, all the methods from the trait are shown in the class as well.

Showing all the methods can be uncomfortable in classes with many methods. For example, the class `Behavior` which uses the trait `TBehavior`, that contains more than 100 methods Figure 7.2. Imagine if we want to add more traits to the composition of `Behavior`, scrolling through all the methods can be uncomfortable.

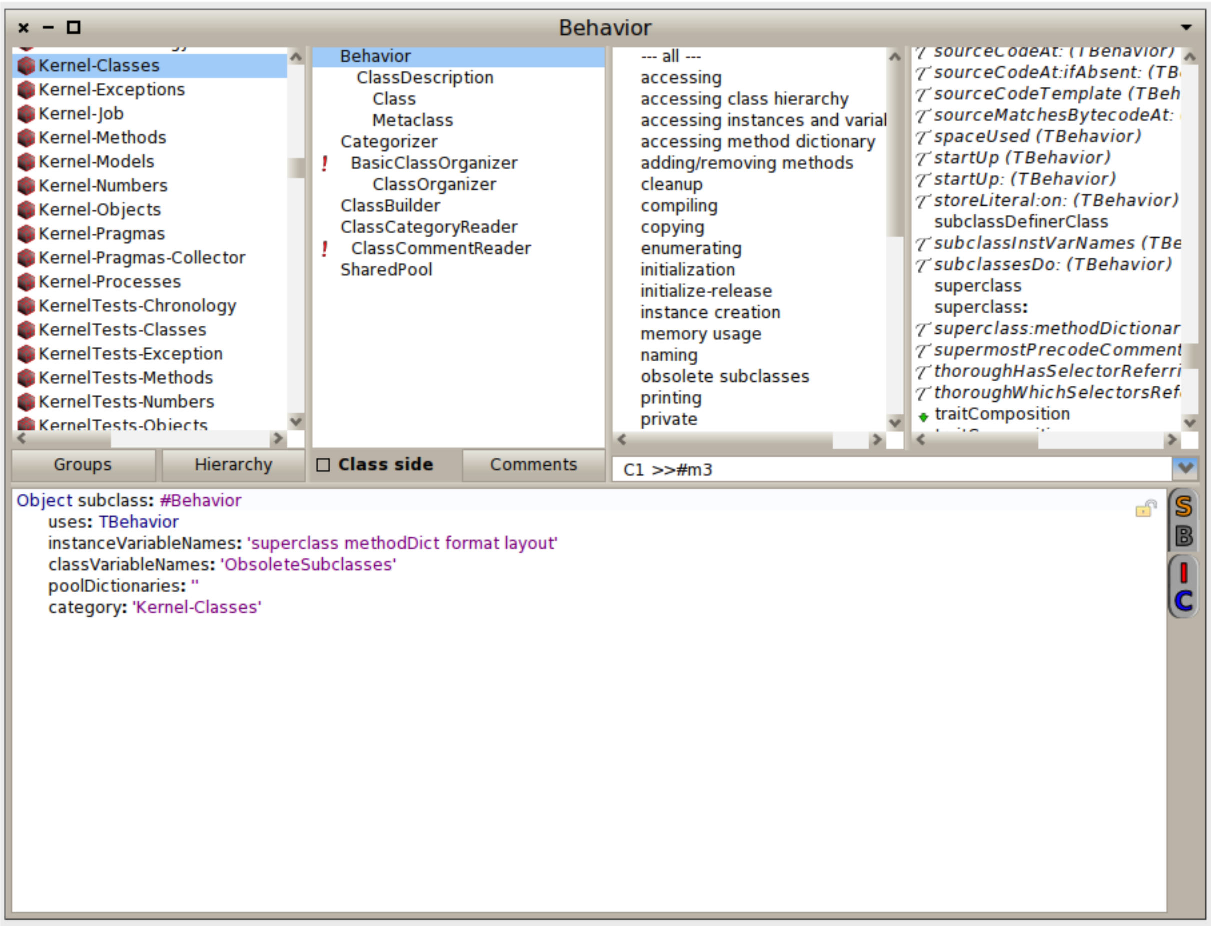


Fig. 7.2: Nautilus

A possible solution could be to filter all the methods that are coming from traits and showing only the local selectors, but this brings problems as well.

In classes with complex trait compositions may be annoying to the programmer by trying to figure out which messages the class knows how to answer. Let's imagine a trait composition like the one in `LinkedListTest` Figure 7.3, to know which methods are coming from the trait the programmer needs to check in the trait composition which traits are being included, and then check for aliases and exclusions. But this solution was not feasible.

These problems lead us to think, should we create a different browser? Is it possible to improve the current browser in order to handle traits in a better way?

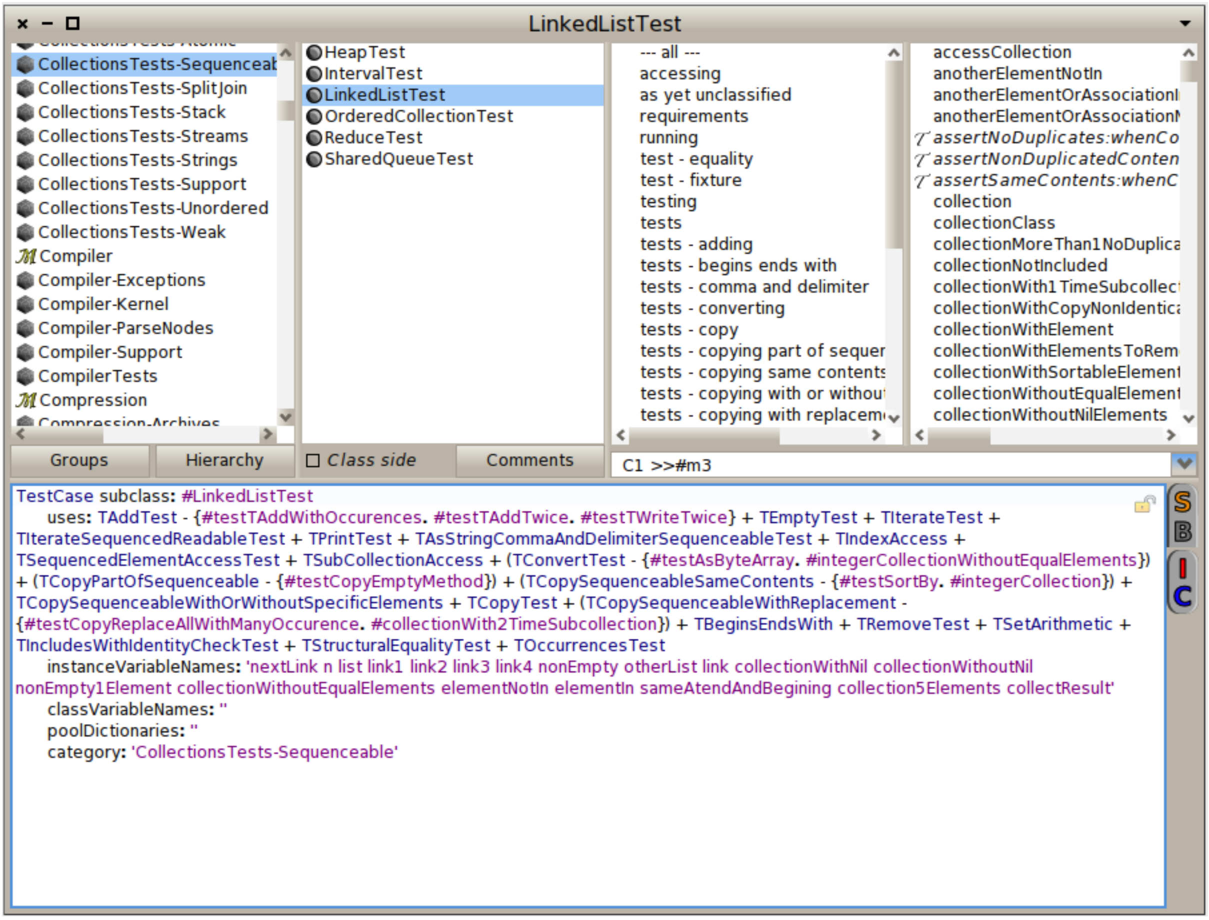


Fig. 7.3: LinkedListTest definition

7.2.3. Thinking ways to visualize traits

The simplest and easiest solution would be to separate in two columns, local methods vs trait methods Figure 7.4, by doing this is possible to differentiate them easily.

The problem with this solution is that you need to scroll two different places in order to look for a method, and this is not comfortable solution. Besides this is not scalable, let's suppose that now someone decides that would be practical to see superclass methods too. Then with this solution you would need to add a third column, turning the browser more complicated and esthetically ugly.

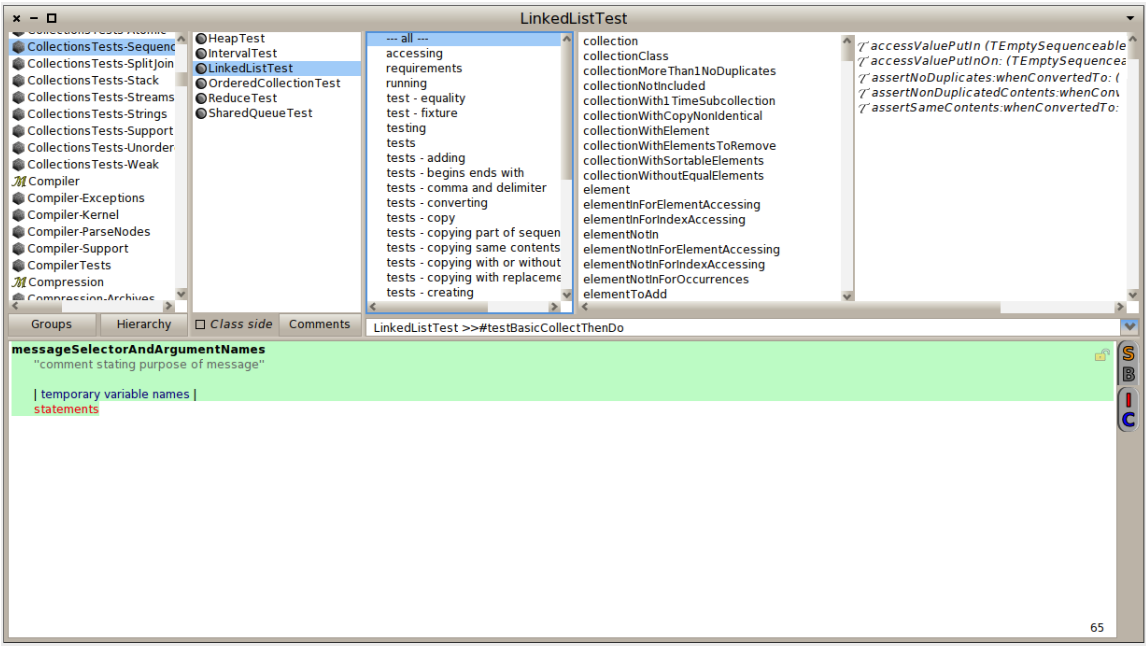


Fig. 7.4: Local methods vs trait methods

Using checkboxes can be a different solution. When a class is selected, we can use a checkbox input to select which methods we want to see: local methods, trait methods and/or superclass methods Figure 7.5. This solution has more uniformity and filtering, it is not a bad solution but it does not look scalable, more checkboxs you add, uglier the browser becomes.

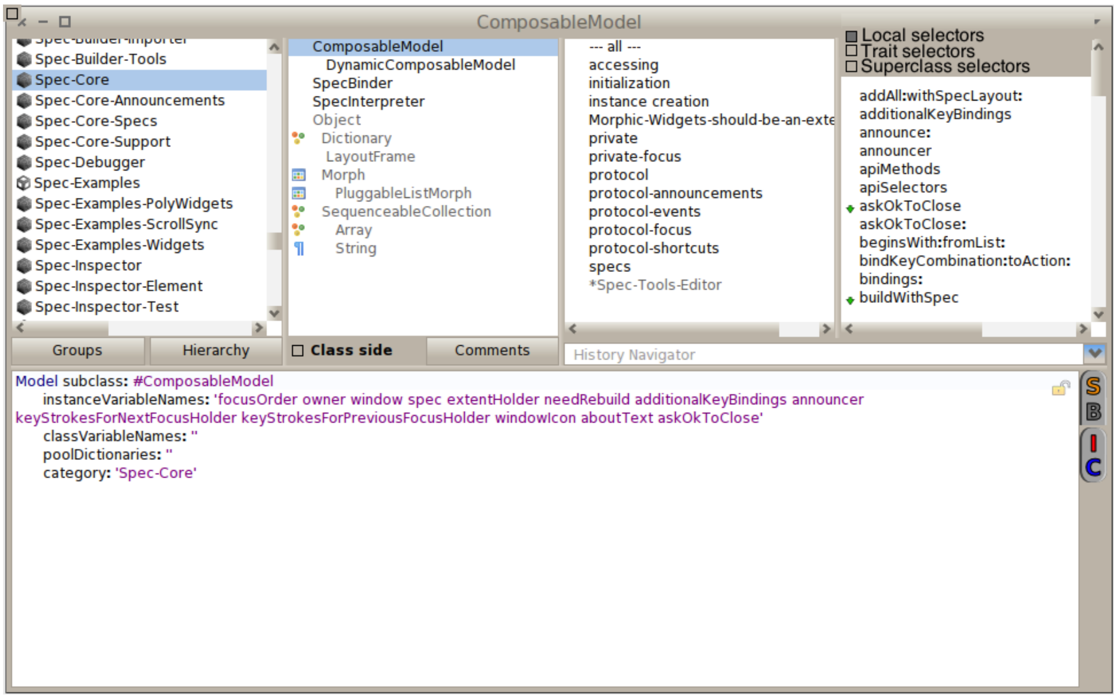


Fig. 7.5: Browser using checkboxes

No browser

In his PhD thesis, Fernando Olivero proposes Gaucho [16], a tool to visualize objects in a different way, without having a browser. The idea is to have “Bubbles” to represent objects in the system, either methods, classes, packages, etc.

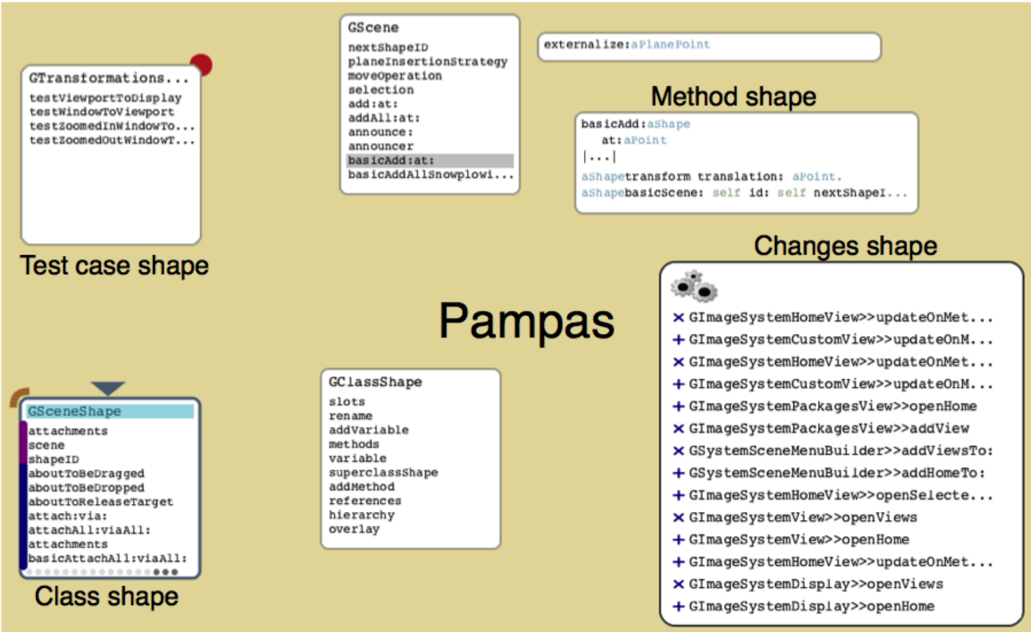


Fig. 7.6: Gaucho

Let’s suppose there is a class `aClass` which have two instance variables, two methods and two traits, then there is a bubble that represents the object `aClass` which shows its instance variables, its methods, its traits, etc. When traits is selected it appears a bubble which shows a list of all the traits of that class, and then it is possible to select one of those traits. Then, once the trait is selected you see all the methods Figure 7.7.

This solution does not have the problems of the previous solutions, it is scalable and the interpretation of the methods in a trait is unique.

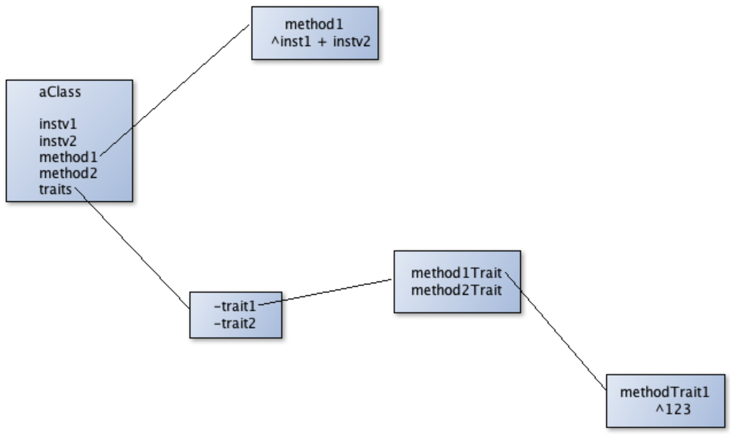


Fig. 7.7: Exploring `aClass` using bubbles

But there is a problem with this solution. Since the old Smalltalk-80 there have been browsers, users are used to work with browsers and changing the way of work can be rejected by the community and difficult to accept. Is it possible to create a browser using this idea of bubbles?

Turning the previous solution into a browser

Taking the same idea of the previous solution, it is possible to change the browser in order to support more features that it had been supporting. Instead of having bubbles, there are columns that can be scrollable.

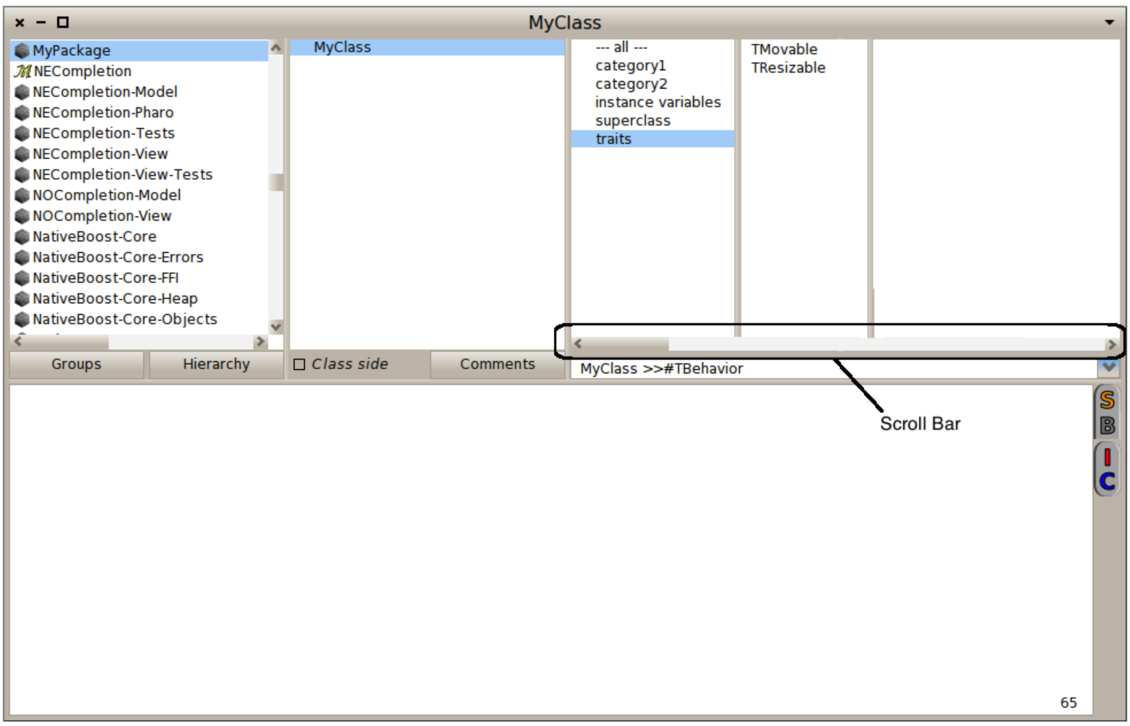


Fig. 7.8: Bubbles turned into a browser

A possible implementation for this browser can be made in `Glamour` [17]. `Glamour` is a dedicated framework to describe the navigation flow of browsers. Thanks to its declarative language, `Glamour` allows one to quickly define new browsers for their data.

`Glamour` also allows to show different presentations of the same pane by using tabs as you can see in figure 7.9. This can be very useful to show a complex hierarchy of classes and traits.

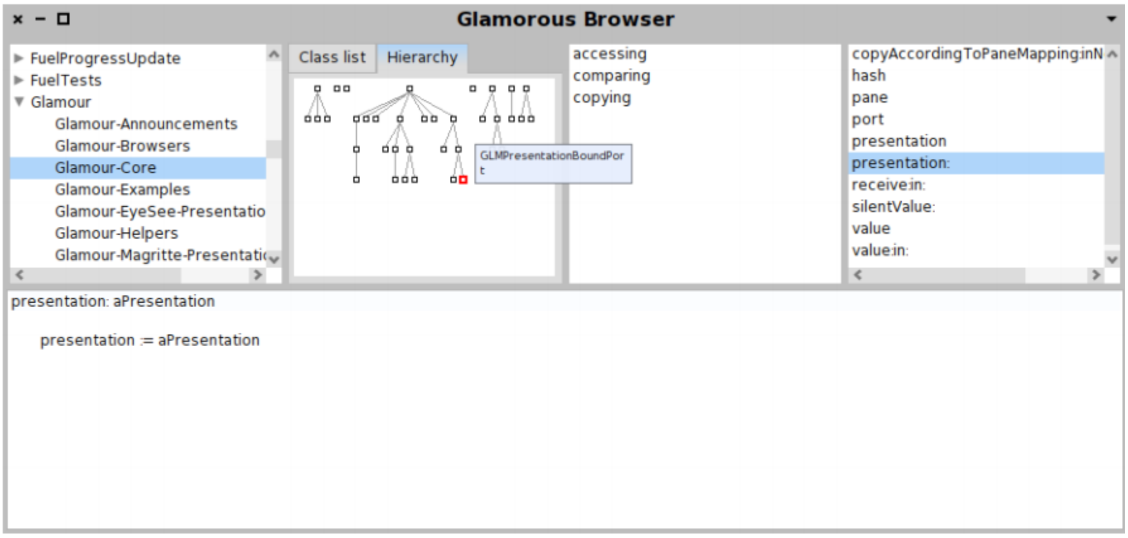


Fig. 7.9: Showing a hierarchy of classes with Glamour

A possible problem with this browser is the scrollbar. In the same way that it is uncomfortable to scroll methods, it will be uncomfortable to scroll traits or classes or other hierarchies. If we have a trait `T1` that uses a trait `T2`, `T2` uses `T3` and so on until `Tn` we will have the same problem as having a large number of methods. Fortunately it is not common to have this kind of hierarchies.

Another possibility for a browser is like the one you can see in figure 7.10.

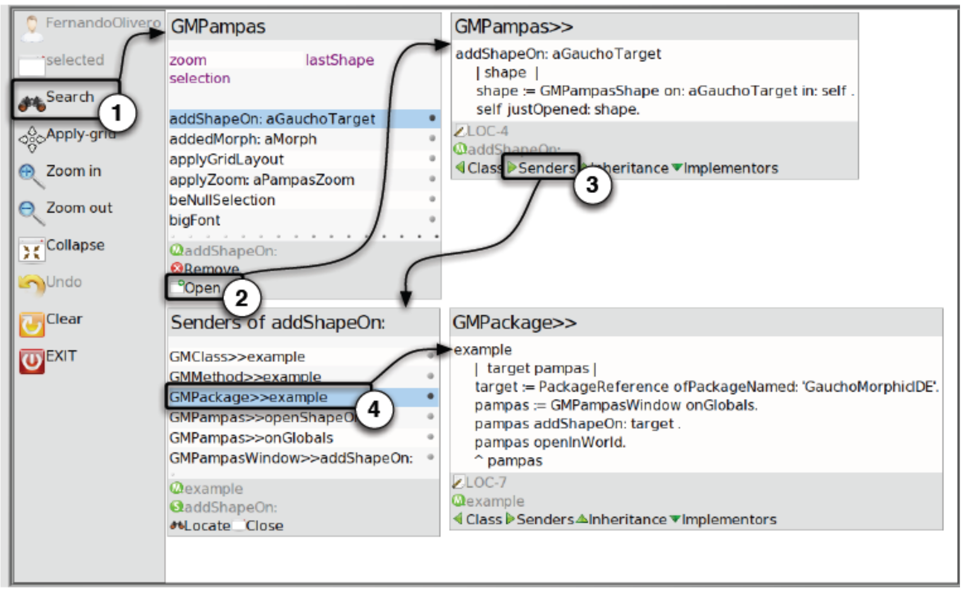


Fig. 7.10: An alternative browser

This is not the typical browser present in Smalltalk, it is more related with the idea of the bubbles. In this example the user does the following steps.

- uses the search widget in the tool-bar to locate the GMPampas class
- scrolls the methods list and select the method `addShapeOn:` and presses the open button
- presses the senders button to open the senders group
- selects the `example` method

The good feature of this last browser is that there is no scrollbar, it is more comfortable to browse, the only limit is given by the size of the screen. The problem arises at the implementation time, it is more difficult to implement since `Glamour` does not support this kind of browsers.

8. CONCLUSIONS

In Chapter 3 we saw how polymorphism between classes and traits was achieved. The goal was to try that tools that know how to handle classes, also know how to handle traits. It is difficult to know if this goal was achieved since tools were not modified. What we did is to put the implementation under test by using the modified system.

The integration process was long, at first we did not know that Monticello had problems when working with kernel classes. As a consequence we had to repeat part of the work we had done and do it again using `change sets`. Even though the integration was successful, it took time and it hindered us to modify tools to test the polymorphism.

As we said, during the importing process all the system worked quite good. There were only a few reported problems but nothing complicated, this led us to think that the implementation was well made and that all the tools that were working before are still working. It will take time to see the polymorphism benefits/disadvantages of the changes.

Regarding the bugs fixed, they also took time. The bug of explicit requirement methods was fixed by modifying the method dictionary depending on how the methods were added. As we commented, this fix was working but it was not well accepted by some of the developers. We had to find another solution, and we modified the implementation of the `explicitRequirement` method.

This fix was useful for myself when we added rules to keep clean code in Pharo. We added code (not related with traits) for checking that methods in classes were not repeated in their superclasses. This bug forced programmers to repeat code so fix it was important.

Regarding the bug in trait compositions, we did not receive complaints of the users. This can be because traits are not widely used, they are a relative new concept. What we know is that after the integration of the fix, the system was heavily used without any problems.

Finally, regarding future work, we tried to dedicate some time thinking about it. The traits visualizer is a needed tool in Pharo, during all this work we realized that the current way of showing traits is not comfortable. Classes with many methods, that use traits with many methods too are difficult to work with. For this reason a trait visualizer would be worth of doing.

9. APPENDIX

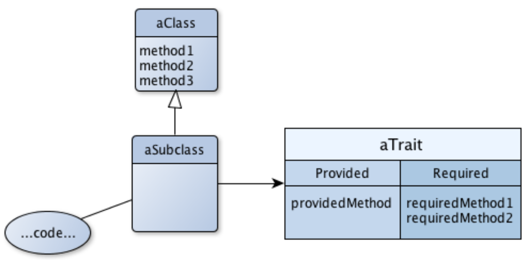
In this section we will detail the graphic notation used in this work.

9.1. Diagram of classes and traits

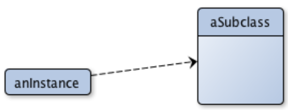
Each box represents either a class or trait. Traits boxes have two columns, the first is to show the provided methods and the second to show the required methods. The circle is only for showing code.

The arrow that goes from `aSubclass` to `aClass` is to represent inheritance. It means that `aSubclass` inherits from `aClass`.

The arrow that goes from `aSubclass` to `aTrait` is to represent that `aSubclass` is using the trait `aTrait`.



In the next figure we can see a box `anInstance` which represents an object that it is instance of `aSubclass`. The dashed arrow is to represent `instanceOf`



9.2. Diagram of sequence

The following figure represents diagram of sequence. It is a representation of sending messages.

- Each box represents an object;
- The time advance from top to bottom;
- Each arrow represents the message sending from one object to another. Dashed arrows represent the `return`;

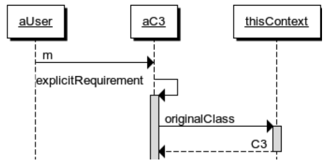


Fig. 9.1:

9.3. Grouping boxes

Finally, there are grouping of boxes (classes, traits, code). The only purpose of this is for clearness.

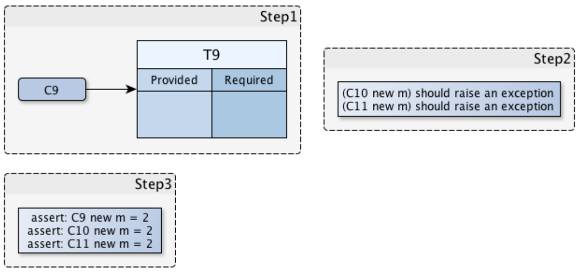


Fig. 9.2:

Bibliografía

[1] Java web site. `http://www.java.com/`.

[2] C++ web site. `http://www.cplusplus.com/`.

[3] Ruby web site. `https://www.ruby-lang.org`.

[4] Nathanael Scharli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. 2003.

[5] Php web site. `http://php.net/`.

[6] Perl web site. `http://www.perl.org/`.

[7] Pharo. `http://www.pharo-project.org/`.

[8] Smalltalk language. `http://www.smalltalk.org/`.

[9] Gilad Bracha and William Cook. Mixin-based inheritance. *SIGPLAN*, 1990.

[10] Alejandro González. Mejorando la utilidad de las herramientas de refactorings. Master's thesis, University of Buenos Aires, 2008.

[11] Squeak web page. `http://www.squeak.org/`.

[12] Mit licence. `http://opensource.org/licenses/mit-license.php`.

[13] Roassal web page. `http://www.moosetechnology.org/tools/roassal`.

[14] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley, 2002.

[15] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *ECOOP*, 1997.

[16] Fernando Olivero, Michele Lanza, and Mircea Lungu. Gaucho: From integrated development environments to direct manipulation environments. 2010.

[17] Glamour. `http://www.moosetechnology.org/tools/glamour`.

[18] A. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. Pharo by example. `http://pharobyexample.org`, 2009.

[19] A. Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. Deep into pharo. `http://pharobyexample.org`, 2009.

[20] S. Ducasse, Oscar Nierstrasz, N. Scharli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 2006.

[21] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Ring: a unifying meta-model and infrastructure for smalltalk source code analysis tools. 2011.

[22] D. Cassou, S. Ducasse, and R. Wuyts. Traits at work: the design of a new trait-based stream library. *Journal of Computer Languages, Systems and Structures*, 2009.

[23] T. Van Cutsem and M.S. Miller. Traits.js: Robust object composition and high-integrity objects for ecmascript 5.

[24] Yed graph editor. `http://www.yworks.com/`.

[25] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the design of existing code*. Addison-Wesley, 1999.

[26] Adele Goldberg and David Robson. *Smalltalk-80: The language and its Implementation*. Addison-Wesley, 1983.

[27] Fernando Olivero. *Object-focused Environments Revisited*. PhD thesis, Faculty of Informatics of Universita della Svizzera Italiana, 2013.

[28] Multiple inheritance. `http://en.wikipedia.org/wiki/Multiple_inheritance`.

[29] Refactorings. `http://es.wikipedia.org/wiki/Refactorizacion`.